Learning Embeddings for Graphs and Other High Dimensional Data

Hasti Narimanzadeh

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 31.12.2019

Thesis supervisor and advisor:

Prof. Parinya Chalermsook

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Hasti Narimanzadeh

Title: Learning Embeddings for Graphs and Other High Dimensional Data

Date: 31.12.2019 Language: English Number of pages: 5+54

Computer Science

Professorship: Theoretical Computer Science

Supervisor and advisor: Prof. Parinya Chalermsook

An immense amount of data is nowadays produced on a daily basis and extracting knowledge from such data proves fruitful for many scientific purposes. Machine learning algorithms are means to such end and have morphed from a nascent research field to omnipresent algorithms running in the background of many applications we use on a daily basis. Low-dimensionality of data, however, is highly conducive to efficient machine learning methods. However, real-world data is seldom low-dimensional; on the contrary, real-world data can be starkly high-dimensional. Such high-dimensional data is exemplified by graph-structured data, such as biological networks of protein-protein interaction, social networks, etc., on which machine learning techniques in their traditional form cannot easily be applied.

The focus of this report is thus to explore algorithms whose aim is to generate representation vectors that best encode structural information of the vertices of graphs. The vectors can be in turn passed onto down-stream machine learning algorithms to classify nodes or predict links among them. This study is firstly prefaced by introducing dimensionality reduction techniques for data residing in geometric spaces, followed by two techniques for embedding vertices of graphs into low-dimensional spaces.

Keywords: Graph embeddings, Network embedding, Machine learning, dimensionality reduction algorithms, random walks, spectral techniques

Preface

Like most things, I would like to keep this preface minimal. So, first and foremost, I would like to thank my supervisor and advisor Parinya Chalermsook for his support throughout, and for granting me the autonomy to do research on my topic of interest. As I recall, he once told me nothing repels him more than unmotivated students; I hope I haven't let him down in that respect.

The friendships made, I will treasure always. So big thanks to Wanchote, Sorrachai, Amit, Ly, and Nidia.

My best friends and partners in crime, Armin and Mona, who also happen to be my brother and sister, I am indebted to you for your guidance and support.

The process would have been a lot more difficult if it wasn't for both emotional and intellectual support, as well as the late night dinners and orange juices Arash provided every single day. Arash, you simply make life better, thank you.

This wouldn't be possible after all without my parents giving me the gift of life. Nader and Shahrbanou, thank you for every step of the way. I will be beholden to you always.

Otaniemi, 31.12.2019

Hasti Narimanzadeh

Contents

1 Introduction And Outline Of The Thesis

Machine Learning (ML) embodies the conversion of experience to knowledge in the realm of computers. The term originates from computers programmed to "learn" from the input data available to them. Machine learning is where the sheer abundance of digital data intersects with mathematical frameworks. In loose terms, the input to such a learning model is the training data and the output is some expertise.

Such pattern recognition in data has a long history: The astronomical observations of Tycho Brahe in the 16th century allowed Kepler to discover the empirical laws of planetary motion, which in turn paved the way for developments in classical mechanics [1]. And nowadays its applications are innumerable, ranging from recommendation systems to image classification. Moreover, the type of data used in such models runs the gamut: images, words, numbers, clicks on social media, etc., enabling all the more applications to be born.

The nature of the input data can speak to a machine learning model's efficacy. Conventionally, data used by learning algorithms are represented in Euclidean space. However, in many real-world scenarios, the data are represented as graphs with complex relationships between its entities. Examples of such data are social networks or biological protein-protein networks [2]. Attempts to apply machine learning algorithms established the area of *representation learning* on graphs whose goal is to encode graphs in low-dimensional embeddings such that the graph structure and properties are maximally preserved. The encoded graph-structured data can be in turn fed into a downstream machine learning algorithm as feature inputs.

Traditionally, representation vectors were hand-crafted for graphs, discarding the graph properties encoded in the graph structure. With the advancement of machine learning paradigms, research gravitated towards applying machine learning algorithms on graphs. Fast forward to now, techniques in representation learning on graphs, also referred to as *graph embedding*, are broadly categorized into three categories of factorization based, random walk based, and deep learning approaches. These categories include countless algorithms and there is ongoing research to concoct all the more with more efficiency, as well as construct a consistent theoretical foundation for all techniques.

In Section 2 we explore the mathematical bedrock of machine learning algorithms and subsequently study a well-known dimensionality reduction technique on data in geometric spaces and how it bears resemblance to a factorization based graph embedding algorithm we study in Section 4. In Section 3 we review the general idea of graph embeddings and provide a literature review of some existing techniques. In Section 5, a random walk embedding technique is explored, and lastly some challenges that lie ahead are discussed.

2 Machine Learning

We begin to study how learning is mathematically defined in the context of machine learning. In so doing, we lay groundwork and justification for numerous elements of the subsequent algorithms and concepts. It can also be viewed as an attempt to bring the two areas of graph embeddings and machine learning closer.

2.1 Notation

The following notations are used throughout this manuscript:

A scalar is denoted by a roman (non-bold) lowercase letter, e.g. $b \in \mathbb{R}$. We denote vectors in $\mathbb{R}^d$ by bold lowercase letters, e.g. $\mathbf{x} = (x_1, x_2, \ldots, x_d)^T$ is a column vector whose elements are $x_1, x_2, \ldots, x_d$ (denoted by roman, indexed lowercase letter) and where $\square^T$ denotes the transpose operator. Norm function $\|\mathbf{x}\|$ without any indices refers to L2-norm:

$$\|\mathbf{x}\| = \|\mathbf{x}\|_2 = \sqrt{\sum_i x_i^2}$$

A bold uppercase letter signifies a matrix, for example $\mathbf{M} \in \mathbb{R}^{d \times d}$ is a square matrix of dimensions d by d. Elements of matrix $\mathbf{M}$ are denoted using roman, capital, double-indexed letters such as M_{ij}

Both $\mathbb{P}$ and Pr may donate the probability of a random variable. The dot product of two vectors is indicated by a dot character, e.g. $\mathbf{w}.\mathbf{x}$ is the dot product of vectors $\mathbf{w}$ and $\mathbf{x}$.

2.2 A Formal Mathematical Learning Model

Learning has a concrete mathematical notion that is essential to understanding how learning algorithms operate. To this end, we attempt to present an all-encompassing albeit terse description of a formal model that describes a simple learning scenario.

Shalev-Shwartz and Ben-David [3] offer a simple but powerful example of a learning problem which we can deliberately examine to seek the primary components of a learner. Suppose we have a few papayas at our disposal and would like to determine their tastiness. Drawing on prior experience we decide to use two features of papayas to assess tastiness: their color and softness. We begin tasting papayas, noting their tastiness, color, and softness. Ultimately, we would have a sample of tasted papayas, having recorded their two features of softness and color. We then arrive at a prediction rule for predicting the tastiness of future papayas.

We begin with the fundamental mathematical components of the said learning problem.

- **Learner's input**: Learner is given the following in a basic statistical setting:
  - **Domain set**: Often denoted by $\mathcal{X}$, it is the set of objects under study that we wish to label. In the above-mentioned learning task, the domain

set is all the papayas. Each object of the domain set is often referred to as *data points* or *instances* and is usually represented as a vector of *features* or *dimensions*.

- **Label set**: This is a set, often denoted by $\mathcal{Y}$, including all the labels we wish to assign to each data point in the domain set. For our example, the label set is thus tasty and not tasty which we can mathematically formulate as either $\{-1, +1\}$ or $\{1, 0\}$. In this specific type of learning scheme, we have the "correct" label for each papaya in the domain set. Nonetheless, this may not always be the case. In fact, learning tasks to which labels are not available are dubbed *unsupervised learning*; whereas learning schemes that exploit existing correct labels to tune their parameters are named *supervised learning*. The most notable example of the former is clustering [4]. And the papaya example is an instance of supervised learning, more specifically, a classification task.

$\mathcal{X}$ and $\mathcal{Y}$ coupled together form a set, namely the *training data set* $\mathcal{S} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, where y_i is the correct label corresponding to data point $\mathbf{x}_i$. Usually, the domain and label sets are fed to learner as the this training data set.

- **Learner's output**: The learner is expected to learn a prediction rule, or better yet a mapping from the domain set to the label set $h : \mathcal{X} \longrightarrow \mathcal{Y}$, which also constitutes the output of the learner. The mapping is also called the *hypothesis* or the *predictor*. The hypothesis is the rule that the learner adopts to assess the tastiness of future papayas based on their color and softness. The correct label y_i is however generated by a function f, that is $f(\mathbf{x}_i)$, where $f : \mathcal{X} \longrightarrow \mathcal{Y}$. The "correct" labelling function f is what the learner tries to emulate to the best of its abilities.

- **The probability distribution of the data** The training data set is sampled from an arbitrary probability distribution. In our current learning scheme, we assume the learner does not have any knowledge of the underlying distribution $\mathcal{D}$ and does not try to estimate it. We will encounter a specific type of learning paradigm in Section 2.5, that the estimation of the underlying probability distribution of the generated data lies at the heart of the learning task. This is however not the case here.

- **Measure of performance** The efficacy of the trained learner is determined by how the learner performs on a random, or rather unseen data point drawn from the underlying distribution. In that sense, the *true error* of the hypothesis h is defined to be the probability of label of a random data point—drawn from probability distribution $\mathcal{D}$—not being equal to the correct label. More precisely,

$$\mathcal{L}_{\mathcal{D},f}(h) \triangleq \mathbb{P}_{x\sim\mathcal{D}}[h(x) \neq f(x)] \tag{1}$$

This measure determines the *generalization* power of the learning algorithm and for this reason it is also called the *generalization error* of the learner. The two terms, "generalization error" and the "true error " may be used interchangeably throughout.

The generalization error is measured with respect to the underlying distribution $\mathcal{D}$ and the correct labelling function f, hence the subscript $(\mathcal{D}, f)$. However, the learner has no knowledge of the two and therefore is incapable of directly calculating the true error. This lack of knowledge about the target function and the distribution of the data underpins the necessity of a different notion of error which the hypothesis h can compute. This brings us to the following learning paradigm: *Empirical Risk Minimization*.

2.3 Empirical Risk Minimization

The learner only has access to the training data set drawn from an unknown distribution. It attempts to learn a hypothesis $h : \mathcal{X} \longrightarrow \mathcal{Y}$ that approximates a target function f that has generated the labels of the training samples. Since the learner has no knowledge of either f or $\mathcal{D}$ and has the training data set at its disposal, it computes the *training error* it incurs on the training samples. In the simplest of forms, such as the papaya example we've discussed so far, this error can be calculated as follows.

$$\mathcal{L}_{\mathcal{S}}(h) = \frac{|\{i \in [N] : h(x_i) \neq f(x_i)\}|}{N} \tag{2}$$

The subscript emphasizes the dependence of the hypothesis loss on the training data set. And due to this dependence, it is also called *empirical risk* or *empirical error*. Intuitively, the hypothesis attempts to do well on the training data set, or rather, minimize the empirical risk and for this reason, this learning paradigm is named *Empirical Risk Minimization* or ERM for short.

2.3.1 Empirical Risk Minimization and Overfitting

Albeit ERM seems intuitive, Shalev-Shwartz and Ben-David [3] outline how it can lead to *overfitting*. Overfitting is the phenomenon that the hypothesis h performs well on the training data set while performing poorly on the unseen data, concluding that the generalization error is too high for such a hypothesis. A common solution to this overfitting problem is to constrict the search space. Strictly speaking, the learner chooses a set of predictors, namely the *hypothesis class* denoted by $\mathcal{H}$, before seeing the training data set. Thereafter the problem amounts to finding a mapping function $h \in \mathcal{H}$ that performs well enough on the training data set. Therefore, formally, we would like to find the set of hypotheses in $\mathcal{H}$ that achieve the minimum empirical error

$$\mathrm{ERM}_{\mathcal{H}}(\mathcal{S}) \in \arg\min_{h \in \mathcal{H}} \mathcal{L}_{\mathcal{S}}(h)$$

The choice of the hypothesis class is thus based on some prior knowledge about the task at hand and leads to a fundamental question in learning theory of how to choose the hypothesis class $\mathcal{H}$ to ensure $\text{ERM}_{\mathcal{H}}$ learning will not overfit. One solution to preventing overfitting is restricting the hypothesis class by imposing an upper bound on its size—which is the number of hypotheses in the hypothesis class. Theorem (1) states that ERM will not overfit provided the hypothesis space $\mathcal{H}$ is finite and the training sample is sufficiently large.

Theorem (1) makes two simplifying assumptions: the *i.i.d.* Assumption and the *realizability* Assumption. The i.i.d. assumption, on which numerous machine learning algorithms are founded, states that data points in the training data set are independently and identically distributed. The realizability assumption is that there exists $h^* \in \mathcal{H}$ such that $\mathcal{L}_{\mathcal{D},f}(h^*) = 0$. Meaning that for a random data point x sampled from $\mathcal{D}$, the true label given by $f(x)$ equals the calculated label $h^*(x)$ with probability 1. Therefore, Theorem (1) must show that

$$\mathcal{L}_{\mathcal{D}}(h_{\mathcal{S}}) < \epsilon \qquad \text{with probability at least } 1 - \delta$$

for *accuracy parameter* $\epsilon > 0$ and *confidence parameter* $1 - \delta$.

That is, the *true* error of the hypothesis chosen by the ERM is sufficiently low, that is $\mathcal{L}_{\mathcal{D}}(h) < \epsilon$. The confidence parameter is to cater for cases that we may draw "unrepresentative" samples from the underlying distribution.

Theorem 1 (FINITE theorem) *Let $\mathcal{H}$ be a finite hypothesis space and assume realizability. Let ϵ and $\delta \in (0, 1)$ be the accuracy and the confidence respectively. And consider the sample size N to be*

$$N \geq \frac{\log\left(|\mathcal{H}|/\delta\right)}{\epsilon}$$

*Let the **Empirical Risk Minimization** algorithm select the hypothesis h_s over the sample $S \sim \mathcal{D}^N$. Then*

$$\mathcal{L}_{\mathcal{D}}(h_s) \leq \epsilon$$

with probability at least $1 - \delta$

Proof: Define $\mathcal{H}_\epsilon = \{h \in \mathcal{H} \colon \mathcal{L}_{\mathcal{D}}(h) > \epsilon\}$, which includes all hypotheses with error more than $\epsilon > 0$. We then move on to calculate the probability that any hypothesis in $\mathcal{H}_\epsilon$ is consistent with a sample S of size N

$$
\begin{aligned}
&\mathbb{P}[\exists h \in \mathcal{H}_\epsilon \colon \mathcal{L}_\mathcal{S}(h) = 0] \\
&= \mathbb{P}[\mathcal{L}_S(h_1) = 0 \vee \ldots \vee \mathcal{L}_\mathcal{S}(h_{|\mathcal{H}_\epsilon|}) = 0] \\
&\leq \sum_{h \in \mathcal{H}_\epsilon} \mathbb{P}[\mathcal{L}_\mathcal{S}(h) = 0], && \text{(Union bound)}
\end{aligned}
$$

Given we have defined the error of the hypotheses to be

$\mathcal{L}_\mathcal{D}(h) = \mathbb{P}_{x \sim \mathcal{D}}[h(x) \neq f(x)] > \epsilon$

we have

$\mathbb{P}_{x \sim \mathcal{D}}[h(x) = f(x)] < 1 - \epsilon$

therefore

$$
\begin{aligned}
&\leq (1-\epsilon)^N |\mathcal{H}_\epsilon| \\
&\leq (1-\epsilon)^N |\mathcal{H}|
\end{aligned}
$$

Hence, the probability that all the hypotheses that are consistent with sample $\mathcal{S}$ have error at most ϵ is at least $1 - (1-\epsilon)^N |\mathcal{H}|$. We would like to select N such that

$$(1-\epsilon)^N |\mathcal{H}| \leq \delta$$

This yields

$$N \geq \frac{\ln(\frac{1}{\delta}) + \ln|\mathcal{H}|}{-\ln(1-\epsilon)} \geq \frac{\ln(|\mathcal{H}|/\delta)}{\epsilon}$$

■

Therefore, in most machine learning scenarios, a suitable *loss function*, also known as the *cost function*, is defined based on the task at hand and subsequently optimized (minimized) by an optimization method of choice. The choice of the loss function primarily hinges on the types of parameters we would like to predict and also affects the choice of the optimization method.

2.4 The Perceptron Algorithm: A Linear Classifier

We study a simple yet powerful linear classifier, the Perceptron [5], to see how the components of a learning task we set forth earlier play out. We consider a binary classification task, i.e. $\mathcal{Y} = \{-1, +1\}$ and a training data set $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ where $\mathbf{x}_i \in \mathbb{R}^d$ for all i.

For such a training set, we would like to find a linear classifier, that is a hyper-plane, which correctly classifies all the data points. A hyper-plane is characterized by the *normal vector* which is perpendicular to the hyper-plane $\mathbf{w} \in \mathbb{R}^d$ and a *bias* parameter $b \in \mathbb{R}$. Thus, the learning task amounts to finding the most suited normal vector and bias. Let the potential hyper-planes, i.e. linear predictors, be the following set of functions

$$L_d = \{\mathbf{x} \mapsto \mathbf{w}.\mathbf{x} + b : \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}\}$$

For any data point $\mathbf{x}$ there are two options—minus the case where the data point resides on the hyper-plane and thus the dot product of the two equals zero—: the data point can form an acute angle with the normal vector and thus the functions in L_d would output a positive number. Or the data point forms an obtuse angle with the normal vector in which case the output would be negative. In any case it is not hard to see that for the predictor to classify data point $\mathbf{x_i}$ with the corresponding correct label y_i correctly, the following must hold true

$$y_i(\mathbf{w}.\mathbf{x}_i + b) \geq 0$$

In order for the predictor to return a valid -1 or $+1$ label upon receiving a data point $\mathbf{x}$, we apply the sign function to predictors, that is

$$\text{sign}(\mathbf{w}.\mathbf{x} + b) = \begin{cases} +1, & \text{if } \mathbf{w}.\mathbf{x} + b \geq 0. \\ -1, & \text{otherwise.} \end{cases} \tag{3}$$

This brings us to the hypothesis class designed for binary classification tasks, namely the class of *halfspaces*

$$HS_d = \text{sign} \circ L_d = \{\mathbf{x} \mapsto \text{sign}(h_{\mathbf{w},b}(\mathbf{x})) : h_{\mathbf{w},b} \in L_d\}$$

where

$$h_{\mathbf{w},b}(\mathbf{x}) = \mathbf{w}.\mathbf{x} + b$$

The Perceptron is an iterative algorithm that finds a linear predictor in such hypothesis class for a given training data set, using the following loss function upon receiving a data point $\mathbf{x}_i$ and its corresponding correct label $y_i \in \{-1, +1\}$

$$\ell_{(\mathbf{x}_i,y_i)}(\mathbf{w}, b)) = \begin{cases} 0, & \text{if } y_i(\mathbf{w}.\mathbf{x}_i + b) \geq 0. \\ -y_i(\mathbf{w}.\mathbf{x}_i + b), & \text{otherwise.} \end{cases} \tag{4}$$

The total loss function is the summation of the loss incurred on all the data points in the training data set $\mathcal{S} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$

$$\mathcal{L}_{\mathcal{S}}(h_{\mathbf{w},b}) = \sum_{i}^{N} \ell_{(\mathbf{x}_i,y_i)}(h_{\mathbf{w},b})$$

Intuitively, if the predictor classifies the data point correctly, no error should be incurred. However, if not, the learner should be penalized in proportion to how wrong it was. The convexity of such a loss function guarantees a global optimum, which is the holy grail of any machine learning algorithm.

To optimize the above loss function, Perceptron utilizes *stochastic gradient descent* (SGD). Stochastic gradient descent is a variation of *gradient descent* (GD). Both algorithms are unconstrained optimization techniques that aim to minimize a function by iteratively nudging the parameters from their initial values in the opposite direction of the gradient of the loss function, until they converge. The two techniques differ in how they treat the training data set. Gradient descent makes a pass through *all* the

data points for each update of the parameters, whereas stochastic gradient descent performs an update of parameters based on every single single data point.

Implementation of stochastic gradient descent is outlined in Algorithm 1.

Algorithm 1: SGD$(\eta, T, \mathcal{S})$ for minimizing $f(\mathbf{w})$

Input: step size η
number of iterations T
labeled training data set $\mathcal{S}$
Output: function parameters $\mathbf{w}$
Initialize $\mathbf{w}^{(1)} = \mathbf{0}$
for $t = 1, \ldots, T$ **do**
  Draw $(\mathbf{x}, y) \in \mathcal{S}$ randomly
  $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla \ell(\mathbf{w}^{(t)}; (\mathbf{x}, y))$
return $\mathbf{w}^{(T+1)}$;

Parameters η and T ensure the convergence of SGD [3, 1].

The Perceptron uses SGD (with step size $\eta = 1$) to minimize the loss function defined in Equation (4) (as can be seen by the partial derivatives with respect to the weights and bias used in the update steps) and is described in Algorithm 2. Novikoff [6] proves that the Perceptron converges if the data is linearly separable (there exists a linear classifier that classifies all the data points in the training data set correctly). The theorem is stated below in Theorem 2. It is interesting how neither the size of domain set $\mathcal{X}$ nor the dimensionality of the data points affect the bound on number of mistakes the perceptron makes. The Perceptron does not necessarily converge for non-linearly separable data [3].

Theorem 2 (Novikoff, 1962) *Consider data points* $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^d$ *where* $||\mathbf{x}_i|| \leq r$ *for all* $i \in [N]$ *for some* $r > 0$. *We assume there exists* $\rho > 0$ *and* $\mathbf{w}^* \in \mathbb{R}^d$ *such that* $\rho \leq \frac{y_i(\mathbf{w}^* . \mathbf{x})}{||\mathbf{w}^*||}$ *(linear separability) for all* $i \in [N]$. *Then, we can state the following about the number of perceptron updates (i.e. the number of prediction mistakes)* T *when processing* $\mathbf{x}_1, \ldots, \mathbf{x}_N$ *is*

$$T \leq \frac{r^2}{\rho^2}$$

The Perceptron is one of a plethora of classifiers, linear or otherwise, in the machine learning realm. It is, however, one of the most simple classification algorithms that also lays the groundwork for a more competent set of classifiers known as the *support vector machines* [3, 1]. The Perceptron is also a precursor to a different learning paradigm known as *artificial neural networks*, that will be briefly discussed in Subsection 2.6.

2.5 Generative Models

When following a *discriminative* approach, such as the Perceptron algorithm in Section 2.4, we are oblivious to the underlying distribution of the data. In other

Algorithm 2: perceptron($\mathcal{S}, T$)

Input: labeled training data set $\mathcal{S}$
number of iterations T
Output: predictor parameters $\mathbf{w}, b$
Initialize $\mathbf{w} = \mathbf{0}$ and $b = 0$
for $t = 1, \ldots, T$ **do**
  if $\exists i$ *s.t.* $y_i(\mathbf{w}^{(t)}.\mathbf{x}_i + b^{(t)}) < 0$ **then**
    $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + y_i\mathbf{x}_i$
    $b^{(t+1)} = b^{(t)} + y_i$
  else
    return $\mathbf{w}^{(T+1)}, b^{(T+1)}$;

words, we do not make any assumptions about the underlying distribution of which the observed data are sampled from and do not attempt to learn it, rather we learn a predictor. However, in *generative models*, we assume the data points are drawn from a specific parametric distribution whose parameters we attempt to estimate.

More specifically, the distribution of the data, the model, has a set of parameters θ which the goal is to learn having observed data set $\mathcal{S}$. In other words, we assume a *distribution family* for the observed data and subsequently estimate its parameters. As an example, if we assume the data is sampled from a Gaussian distribution, the parameters are the mean and the covariance, $\theta = \{\mu, \Sigma\}$. Generally, given the training data set $\mathcal{S}$, we have some *prior* belief about θ, and the *Bayes's rule* will update it in light of newly observed data

$$\mathbb{P}[\theta|\mathcal{S}] = \frac{\mathbb{P}[\mathcal{S}|\theta]\mathbb{P}[\theta]}{\mathbb{P}[\mathcal{S}]} \tag{5}$$

The term $\mathbb{P}[\mathcal{S}|\theta]$ is the likelihood of the observed data given θ and can be viewed as a function of θ. $\mathbb{P}[\theta]$ is called the prior, and $\mathbb{P}[\mathcal{S}]$ the marginal likelihood or the evidence.

A brief description of a well-known statistical method is underway, for estimating the parameters of the distribution over the data, that is the *maximum likelihood principle*. Maximum likelihood aims to maximize the likelihood of the observed data which is, in turn, a function of θ, hence the name. Moreover, we will observe how maximum likelihood is similar to and differs from the empirical risk minimization (ERM) studied previously.

2.5.1 Maximum Likelihood Estimator

Simply put, maximum likelihood estimator finds the parameters of the underlying distribution for which the observed data has the highest probability. More concretely, it maximizes the likelihood function described above. Shalev-Shwartz and Ben-David [3] present a concrete example to grasp the idea of this estimator as follows:

Assume a pharmaceutical company has developed a new medicine as a cure to a deadly disease and would like to gauge the probability of patients' survival having used the medicine. The company samples N people independently and provides them with the medicine. The sample set is denoted by $\mathcal{S} = \{x_1, \ldots, x_N\}$, where $x_i = 1$ if patient i survived and $x_i = 0$ otherwise. It can be seen that the random variable x_i follows a Bernoulli distribution which can be modeled with a single parameter $\theta \in [0, 1]$, indicating the probability of survival.

The goal is now to estimate θ based on the training data set $\mathcal{S}$. The joint probability distribution over i.i.d. random variables is

$$\mathbb{P}[x_i, \ldots, x_N|\theta] = \prod_{i=1}^{N} \mathbb{P}[x_i|\theta]$$

Therefore, the probability of generating the data set $\mathcal{D}$ for the given example is

$$\mathbb{P}[D = \{x_i, \ldots, x_N\}] = \prod_{i=1}^{N} \theta^{x_i}(1-\theta)^{1-x_i} = \theta^{\sum_i x_i}(1-\theta)^{\sum_i (1-x_i)} \tag{6}$$

It is often more convenient to work with the logarithm of the likelihood function. Since logarithm is a monotonic function, the maxima of the logarithm of the likelihood function, *log-likelihood* for short, will coincide with that of the likelihood function. Let $L(\mathcal{S};\theta)$ denote the log-likelihood of $\mathcal{S}$ given θ, which is

$$L(\mathcal{S};\theta) = \log(\mathbb{P}[\mathcal{S} = \{x_1, \ldots, x_N\}]) = \log(\theta)\sum_i x_i + \log(1-\theta)\sum_i (1-x_i)$$

The maximum likelihood estimation is then the parameter $\hat{\theta}$ such that it maximizes the log-likelihood above

$$\hat{\theta} \in \arg\max_{\theta} L(\mathcal{S};\theta) \tag{7}$$

To find $\hat{\theta}$ that maximizes the log-likelihood, we take the derivative of $L(\mathcal{D};\theta)$ with respect to θ and set it to zero

$$\frac{d}{d\theta} L(\mathcal{S};\theta) = 0$$

Taking the derivative of the log-likelihood of our problem, we can arrive at

$$\frac{\sum_i x_i}{\theta} - \frac{\sum_i (1-x_i)}{1-\theta} = 0$$

Solving this equation we calculate the estimated parameter $\hat{\theta}$ to be

$$\hat{\theta} = \frac{\sum_{i=1}^{N} x_i}{N}$$

The estimated probability of survival, based on maximum likelihood estimation, is just the fraction of people on the drug who have survived to the total number of the users of the drug, which also happens to be in line with our intuition as to what our best guess as to the probability of survival based on currently known data would be.

In the above-mentioned example the Bernoulli random variable was discrete. A slight modification is in order for continuous random variables. Due to the fact that for a continuous random variable X we have $\mathbb{P}[X = x] = 0$, we define the log-likelihood to be the log of the *probability density function* of X at x. More specifically, given an i.i.d. training data set $\mathcal{S} = \{x_1, \ldots, x_N\}$ sampled from probability distribution $\mathcal{P}_\theta$, the log-likelihood is

$$L(\mathcal{S};\theta) = \log(\prod_{i=1}^{m} \mathcal{P}_\theta(x_i)) = \sum_{i=1}^{m} \log(\mathcal{P}_\theta(x_i))$$

Shalev-Shwartz and Ben-David [3] adopt the notation $\mathcal{P}[X = x]$ to describe the probability of $X = x$ for both discrete and continuous random variables and the same notation will be used throughout.

2.5.1.1 Maximum Likelihood and Empirical Risk Minimization

The central difference between the maximum likelihood estimator and the Empirical Risk Minimization principle is that in the latter there exists an oblivion towards the underlying distribution of the sampled data. That is, we have a hypothesis class $\mathcal{H}$ that includes the potential hypotheses. The training data set is then used to choose the hypothesis $h \in \mathcal{H}$ that minimized the empirical risk.

Shalev-Shwartz and Ben-David [3] show how the maximum likelihood estimator is an ERM for a particular loss function, known as the log-loss.

Assuming we have estimated the parameter of the distribution to be $\hat{\theta}$, we need to define a loss function to assess the quality of the estimation. One sensible candidate is the following loss function

$$\ell(\hat{\theta}, x) = -\log(\mathcal{P}_{\hat{\theta}}[x]) \tag{8}$$

$\ell(\hat{\theta}, x)$ is, thus, the loss incurred on x if it is sampled from the distribution $\mathcal{P}_{\hat{\theta}}$. It stands to reason that if $\mathcal{P}_{\hat{\theta}}[x_i]$, that is the probability of drawing x_i from the distribution $\mathcal{P}_{\hat{\theta}}$ with the estimated parameter $\hat{\theta}$, is low, then the loss should be high. It follows immediately that the maximum likelihood principle is equivalent to minimizing the log-loss function in Equation (8)

$$\arg\max_\theta \sum_{i=1}^{N} \log(\mathcal{P}_\theta[x_i]) = \arg\min_\theta \sum_{i=1}^{N} -\log(\mathcal{P}_\theta[x_i])$$

We assume that the data follows a distribution $\mathcal{P}$ whose parameters do not necessarily conform with the parameters $\hat{\theta}$ we have estimated. Therefore, a measure of the risk of θ must be devised. And in in fact, Shalev-Shwartz and Ben-David [3] define the true risk of θ to be the expected value of the loss on θ defined in Equation (8)

$$\begin{aligned}
\mathbb{E}[\ell(\theta, x)] &= -\sum_{i=1}^{N} \mathcal{P}[x_i] \log(\mathcal{P}_\theta[x_i]) \\
&= \sum_{i=1}^{N} \mathcal{P}[x_i] \log(\frac{\mathcal{P}[x_i]}{\mathcal{P}_\theta[x_i]\mathcal{P}[x_i]}) \\
&= \underbrace{\sum_{i=1}^{N} \mathcal{P}[x_i] \log(\frac{\mathcal{P}[x_i]}{\mathcal{P}_\theta[x_i]})}_{D_{KL}[\mathcal{P}||\mathcal{P}_\theta]} + \underbrace{\sum_{i=1}^{N} \mathcal{P}[x_i] \log(\frac{1}{\mathcal{P}[x_i]})}_{H(\mathcal{P})}
\end{aligned} \tag{9}$$

The terms D_{KL} and $H(\mathcal{P})$ are *Kullback–Leibler divergence* (or the *relative entropy* [3]) and *entropy* respectively. The former is a measure of difference (divergence) between two probability distributions and the latter a measure of how unpredictablilty of a certain probability distribution. It is noteworthy that that Kullback–Leibler divergence equates zero if the true and the estimated distributions are the same, otherwise equal to a non-negative value for discrete random variables. Also we should note that although frequently introduced as a measure of distance between two probability distributions, Kullback–Leibler divergence is not a metric as it is not necessarily symmetric.

The true risk just computed is the loss function of choice in tasks where we would like to predict a probability, e.g. a classification problem where the output is a probability distribution among all the available labels in the label set $\mathcal{Y}$. We will see this particular loss function in use in Section 5.

2.6 Artificial Neural Networks

Artificial neural networks are computational models that have proved exceptionally powerful in tasks such as image classification, speech recognition, etc. due to their ability to capture non-linearity in data. Different architectures of these brain-inspired networks cater for different tasks, for example, *convolutional neural networks* are widely in use for image classification tasks [7]. Or *recurrent neural networks* for speech recognition [8]. We will briefly review the most simple neural network architecture, the *feed-forward neural networks*, to set a rough foundation for the subsequent sections.

Feed-forward neural networks are directed acyclic graphs $G = (\mathcal{V}, E)$, with collection of neurons (as nodes) stacked in a layers, and a weight function $w : E \longrightarrow \mathbb{R}$. An example of a feed-forward network is illustrated in Figure 1.

Every neuron computes a weighted sum of all its incoming edges and then applies a function to the computed weighted sum. An example neuron depicted in Figure 2 calculates value h as follows

$$h = \sigma(\sum_{i=1}^{4} x_i w_i + b) = \sigma(\mathbf{x}.\mathbf{w} + b)$$

where $\sigma(.)$ is a non-linear **activation function**, $\mathbf{w} = (w_1, w_2, w_3, w_4)^T$ is a weight vector in correspondence with the incoming edges, and b is the neuron's bias.

The weights of all edges are the parameters to be learned. And the weights of the blue edges are called *bias*. Each neuron has its own bias as depicted in Figure 1. Every neural network is composed of at least one input layer, one output layer and an arbitrary number of hidden layers. The input layer is where a data point is fed into, with each neuron in the layer accounting for all dimensions of the data point. The output layer is tailored according to the task at hand; for instance, if we want to predict a scalar label $y \in \mathbb{R}$, there would be one neuron in the output layer, determining probability of presence of each digit $(0-9)$ of an image of a hand-written digit would require one neuron for each digit. The hidden layers, however, can have arbitrary number of neurons and the term "deep" in deep-learning is associated with

the number of hidden layers in a neural network. It is this existence of hidden layers that enables neural networks to learn non-linearly separable problems.

If we define the activation function of a single neuron to be the sign function we defined in Equation (3), it is not hard to see that this neuron would be an implementation the Perceptron algorithm described in Section 2.4. And for this reason, such neuron is called the "perceptron" neuron in the literature [5]. In practice the choice of the activation function is dependent on the nature of the problem being solved; for example, in multi-class or binary classification tasks where what is being predicted is a probability distribution over the class labels, it is common to use softmax and sigmoid functions respectively to approximate probabilities, due to their smooth and bounded (normalized in case of softmax) outputs [9]. However, these should not be the sole reasons why such functions are used to approximate probabilities; many other differentiable and normalized functions can be viable candidates. Nonetheless, both functions are commonly used in the last layer of neural networks to estimate probability distribution in practice.

Softmax in particular is derived from Boltzmann's distribution, which is the maximum entropy ($\arg\max_p(-\sum_i p_i log_2 p_i)$) probability distribution that a system with temperature T occupies a certain state s_i, where each state has a different energy level ϵ_i and a defined mean energy $\sum_i p_i \epsilon_i$ across all possible states:

$$p_i = \frac{e^{\beta \epsilon_i}}{\sum_j e^{\beta \epsilon_j}}$$

where coldness $\beta = \frac{1}{kT}$ and k is a constant. In many instances, e.g. Equation (38), parameter β is assumed to be equal to 1.

We will observe both softmax and sigmoid functions approximating probabilities in practice in Section 5.

The parameters of neural networks are the weights and biases that need to be learned. Given that the number of parameters are often inextricably large, calculating gradient of the loss function, which is needed for gradient based optimization methods such as stochastic gradient descent (SGD), becomes impractical. For this reason, *back-propagation* is used to calculate partial derivatives of loss function at each layer by going through layers backwards and employing chain rule to eliminate redundant calculations compared to direct calculation of gradient based on the general form of the loss function [10].

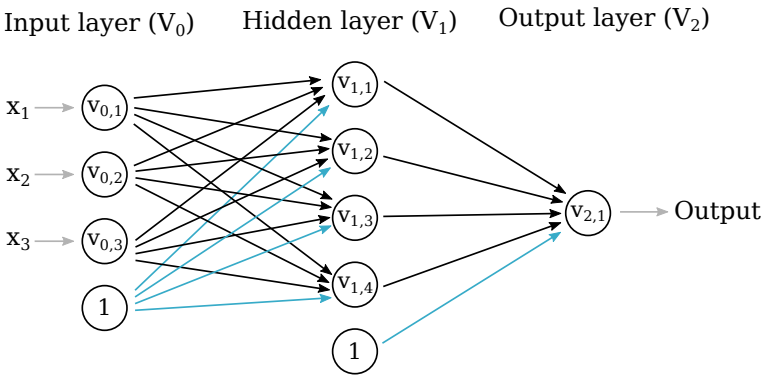

Figure 1: A feed-forward networks with one hidden layer. A data point $\mathbf{x} = (x_1, x_2, x_3)^T$ is fed into the network, for which a single label is produced by neuron $V_{2,1}$. The biases of the neurons correspond to the weights of the blue edges.

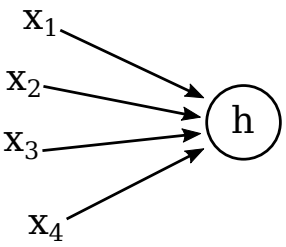

Figure 2: A computational neuron with four incoming edges. The output h is the weighted sum of all the incoming edges $x_1, \ldots, x_4$, passed to some non-linear function.

2.7 Dimensionality Reduction

Embedding high-dimensional graph-structured data into lower-dimensional vector spaces is the centerpiece of this thesis, which is at heart a dimensionality reduction procedure. Before diving into techniques for graph embedding, we study a much revered unsupervised learning algorithm for reducing the dimensionality of Euclidean data, namely the *Principal Component analysis*, PCA for short. We will observe how it bears resemblance to the well-known graph embedding technique *Laplacian eigenmaps* studied in Section 4.

Dimensionality reduction is a technique of mapping high-dimensional data into a low-dimensional space, and so akin to the concept of lossy compression in information theory. In many cases, the high-dimensional data points might lie on a low-dimensional manifold and thus learning such a manifold would yield the said mapping. In fact, the terms 'manifold learning' and 'dimensionality reduction' are often used interchangeably for this reason [11].

There are several reasons to reduce the dimensionality of the data. From a computational efficiency standpoint, manipulating high-dimensional data can be challenging. Furthermore, high-dimensional data can result in substandard generalization ability of the learning algorithms and in that sense, dimensionality reduction is said to have a de-noising effect [3]. More generally, dimensionality reduction can be used for extracting meaningful structures from the data and for visualization purposes.

Many dimensionality reduction algorithms have been developed, of which the perhaps most well-known is the *Principle Component Analysis* (PCA). PCA both compresses and recovers the data points by applying linear transformations and aims to find such transformations such that the differences between the original vectors and the recovered vectors are minimal. PCA can also be viewed in light of variance maximization. More specifically, transformations are such that the variance of the projected data points is maximal.

There are multitudes of algorithms that apply non-linear transformation such as Kernel PCA [12] and Laplacian eigenmaps [13]. The Laplacian eigenmaps will be reviewed extensively in Section 4.

2.7.1 Principle Component Analysis: Reconstruction Error Minimization

As said before, PCA is an unsupervised learning algorithm; that is the label set $\mathcal{Y}$ is an empty set. The domain set $\mathcal{X}$ includes $\mathbf{x}_1, \ldots, \mathbf{x}_N$ that are N column vectors (data points) residing in $\mathbb{R}^d$. PCA aims to apply a linear transformation to reduce the dimensionality of the vectors from d to k where $k < d$. Let matrix $\mathbf{W} \in \mathbb{R}^{k \times d}$ represent the linear transformation which induces the mapping $\mathbf{x} \mapsto \mathbf{W}\mathbf{x}$. So, $\mathbf{W}\mathbf{x}$ is in $\mathbb{R}^k$ and is the lower-dimensional representation of the vector $\mathbf{x}$. Having defined the compression matrix, we define matrix $\mathbf{U} \in \mathbb{R}^{d \times k}$ to be the decompression matrix that recovers vector $\mathbf{x}$ from its compressed version in the k-dimensional linear subspace. That is, the reconstructed vector $\hat{\mathbf{x}}$ is equal to $\mathbf{U}\mathbf{W}\mathbf{x}$ and resides in the high dimensional space $\mathbb{R}^d$.

Having written out the compressed and recovered vectors in mathematical terms, the PCA problem reduces to minimizing the reconstruction error in the least square sense. That is, we wish to find the matrices $\mathbf{W}$ and $\mathbf{U}$ for which the total squared distance between the original and reconstructed vectors among all the data points is minimized (which constitutes the loss function $\mathcal{L}$)

$$\underset{\mathbf{W}\in\mathbb{R}^{k\times d},\mathbf{U}\in\mathbb{R}^{d\times k}}{\arg\min} \sum_{i=1}^{n} ||\mathbf{x}_i - \mathbf{U}\mathbf{W}\mathbf{x}_i||_2^2 \tag{10}$$

Additional to this error minimization perspective there is an alternative way to define PCA. We can also view PCA as finding projections which maximize the variance. More concretely, the first principal component is the vector in the original space along which the projections of the data points have the largest variance. The second principal component is the vector that maximizes the variance along all the vectors orthogonal to the first, and so on. We will see shortly how the variance maximization is mathematically equivalent to minimizing the reconstruction error.

In the following lemma, we see that the solution to the Equation (10) takes a specific form.

Lemma 3 *Let* $\mathbf{W}$ *and* $\mathbf{U}$ *be the optimal solutions to Equation* (10)*. Then* $\mathbf{W} = \mathbf{U}^T$ *and the matrix* $\mathbf{U}$ *is orthogonal; that is* $\mathbf{U}^T\mathbf{U}$ *is the identity matrix* I_k *in* $\mathbb{R}^k$

Proof: For any $\mathbf{W}$ and $\mathbf{U}$, the mapping $\mathbf{x} \mapsto \mathbf{U}\mathbf{W}\mathbf{x}$ has the range R of k dimensions, that is, it is a k dimensional linear subspace of $\mathbb{R}^d$. Such a space then must have a corresponding matrix $\mathbf{V} \in \mathbb{R}^{d\times k}$, whose columns are the orthonormal basis vectors of this subspace—which is the new coordinate system—meaning $\mathbf{V}^T\mathbf{V} = I$, with range R. Thus, any vector in this subspace can be written as $\mathbf{V}\mathbf{z}$ where $\mathbf{z} \in \mathbb{R}^k$. The term $\mathbf{V}\mathbf{z}$ is viewed as the reconstructed vector in the original vector space of dimensional d from the low-dimensional representation vector $\mathbf{z}$. The difference between the reconstructed and original vectors, for every $\mathbf{z} \in \mathbb{R}^k$ and $\mathbf{x} \in \mathbb{R}^d$, can thus be written as follows

$$f(\mathbf{x},\mathbf{z}) = ||\mathbf{x} - \mathbf{V}\mathbf{z}||_2^2 = \mathbf{x}^T\mathbf{x} + \mathbf{z}^T\mathbf{V}^T\mathbf{V}\mathbf{z} - 2\mathbf{z}^T\mathbf{V}^T\mathbf{x} = ||\mathbf{x}||_2^2 + ||\mathbf{z}||_2^2 + -2\mathbf{z}^T\mathbf{V}^T\mathbf{x}$$

Where we used the fact that $\mathbf{V}^T\mathbf{V} = I_k$. To find the vector $\mathbf{z}$ that minimizes the preceding expression, we compute the gradient with respect to $\mathbf{z}$ and set it to zero

$$\nabla_{\mathbf{z}} f(\mathbf{x},\mathbf{z}) = 2\mathbf{z} - 2\mathbf{V}^T\mathbf{x} = 0 \Longrightarrow \mathbf{z} = \mathbf{V}^T\mathbf{x}$$

Therefore, for every data point $\mathbf{x}$ we can write

$$\mathbf{V}\mathbf{V}^T\mathbf{x} = \underset{\hat{\mathbf{x}}\in R}{\arg\min} ||\mathbf{x} - \hat{\mathbf{x}}||$$

This provides a lower bound on the objective function stated in Equation (10). Thus

$$\sum_{i=1}^{n} ||\mathbf{x}_i - \mathbf{U}\mathbf{W}\mathbf{x}_i||_2^2 = \sum_{i=1}^{n} ||\mathbf{x}_i - \mathbf{V}\mathbf{z}_i||_2^2 \geq \sum_{i=1}^{n} ||\mathbf{x}_i - \mathbf{V}\mathbf{V}^T\mathbf{x}_i||_2^2$$

Since this inequality holds for ever $\mathbf{W}$ and $\mathbf{U}$, the optimal solutions for them are $\mathbf{V}^T$ and $\mathbf{V}$, and this concludes the proof. ∎

The optimization problem stated in Equation (10) can thus be re-written as follows.

$$\arg\min_{\mathbf{U}\in\mathbb{R}^{d\times k}, \mathbf{U}^T\mathbf{U}=I} \sum_{i=1}^{n} ||\mathbf{x}_i - \mathbf{U}\mathbf{U}^T\mathbf{x}_i||_2^2 \tag{11}$$

The matrix $\mathbf{U}\mathbf{U}^T$ is the projection matrix whose columns are orthonormal. Specifically, $\mathbf{U}^T\mathbf{x}_i$ is the projection of $\mathbf{x}_i$ onto the subspace spanned by the columns of $\mathbf{U}$. And $\mathbf{U}\mathbf{U}^T\mathbf{x}_i$ is the reconstructed $\mathbf{x}_i$ in the original coordinate system.

We further expand out the term in Equation (11)'s sum for every $\mathbf{x} \in \mathbb{R}^d$ and matrix $\mathbf{U} \in \mathbb{R}^{d\times k}$ such that $\mathbf{U}^T\mathbf{U} = I$ as follows

$$\begin{aligned} ||\mathbf{x} - \mathbf{U}\mathbf{U}^T\mathbf{x}||_2^2 &= \mathbf{x}^T\mathbf{x} + \mathbf{x}^T\mathbf{U}\mathbf{U}^T\mathbf{U}\mathbf{U}^T\mathbf{x} - 2\mathbf{x}^T\mathbf{U}\mathbf{U}^T\mathbf{x} \\ &= ||\mathbf{x}||^2 - \mathbf{x}^T\mathbf{U}\mathbf{U}^T\mathbf{x} \\ &= ||\mathbf{x}||^2 - \text{trace}(\mathbf{x}^T\mathbf{U}\mathbf{U}^T\mathbf{x}) \\ &= ||\mathbf{x}||^2 - \text{trace}(\mathbf{U}^T\mathbf{x}\mathbf{x}^T\mathbf{U}) \end{aligned} \tag{12}$$

Minimizing Equation (12) is equivalent to maximizing the trace term since the length of vectors are given. So, the minimization problem in Equation (11) can be re-written as a maximization problem. Using the fact that trace is a linear operator, we now aim to solve the following optimization problem

$$\arg\max_{\mathbf{U}\in\mathbb{R}^{d\times k}, \mathbf{U}^T\mathbf{U}=I} \text{trace}\left(\mathbf{U}^T \sum_{i=1}^{n} \mathbf{x}_i\mathbf{x}_i^T\mathbf{U}\right) \tag{13}$$

There are at least two ways to solve Equation (13) which are the spectral theorem and the Lagrange multiplier method. The former approach is what follows

Let $\mathbf{M}$ denote the matrix $\sum_{i=1}^{N} \mathbf{x}_i\mathbf{x}_i^T$. The matrix $\mathbf{M}$ is symmetric, as can be seen, and thus allows for the use of the spectral theorem and so can be written using its structural decomposition as $\mathbf{M} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$ where the matrix $\mathbf{V}$ has the orthonormal eigenvectors of $\mathbf{M}$ as its columns which in turn entails that $\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T = \mathbf{V}^T\mathbf{\Lambda}\mathbf{V} = I$ and the corresponding eigenvalues are along the diagonal of the diagonal matrix $\mathbf{\Lambda}$. We also assume that the eigenvalues are ordered in a descending order, that is $\mathbf{\Lambda}_{11} \geq \mathbf{\Lambda}_{22} \geq \ldots \geq \mathbf{\Lambda}_{dd}$. Moreover, $\mathbf{M}$ is positive semidefinite, meaning all its eigenvalues are non-negative. Having established these characteristics, we can study the following theorem that claims the solution to Equation (13), matrix $\mathbf{U}$, takes a specific form.

Theorem 4 *Given arbitrary vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in $\mathbb{R}^d$, let $\mathbf{M} = \sum_{i=1}^{N} \mathbf{x}_i\mathbf{x}_i^T$, and let $\mathbf{U}$ be a matrix whose columns are the eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_k$ of $\mathbf{M}$ corresponding*

to its k largest eigenvalue.Then, the solution to the optimization problem given in Equation (13) is matrix $\mathbf{U}$ *and* $\mathbf{W} = \mathbf{U}^T$

Shalev-Shwartz and Ben-David [3] provide a solid proof using the spectral decomposition theorem:

Proof: We choose an arbitrary matrix $\mathbf{U} \in \mathbb{R}^{d\times k}$ whose columns are orthonormal and let $\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$ be the spectral decomposition of matrix $\mathbf{M}$. Additionally, we define matrix $\mathbf{B}$ to be $\mathbf{V}^T\mathbf{U}$. Then, $\mathbf{V}\mathbf{B} = \mathbf{V}\mathbf{V}^T\mathbf{U}$. Given that matrix $\mathbf{V}$ is an orthogonal matrix we have $\mathbf{V}\mathbf{B} = \mathbf{U}$ and consequently the following equality also holds

$$\mathbf{U}^T\mathbf{M}\mathbf{U} = \mathbf{B}^T\mathbf{V}^T\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T\mathbf{V}\mathbf{B} = \mathbf{B}^T\mathbf{\Lambda}\mathbf{B}$$

Where $\mathbf{B}^T\mathbf{B} = I$, meaning its columns are orthonormal. Then, we can write out the trace of the resulting matrix as

$$\text{trace}(\mathbf{U}^T\mathbf{M}\mathbf{U}) = \sum_{i=1}^{d} \Lambda_{ii} \sum_{j=1}^{k} B_{ij}^2$$

We attempt to exploit a square matrix instead of $\mathbf{B}$ to provide an upper bound on the trace value. Let's define $\hat{\mathbf{B}}$ to be in $\mathbb{R}^{d\times d}$ whose first k columns are that of $\mathbf{B}$ and to also be orthogonal, that is $\hat{\mathbf{B}}^T\hat{\mathbf{B}} = \hat{\mathbf{B}}\hat{\mathbf{B}}^T$. The orthogonality of matrix $\hat{\mathbf{B}}$ allows us to have $\sum_{j=1}^{d} \hat{B}_{ij}^2 = 1$ for every row i, which in turn implies that $\sum_{j=1}^{k} B_{ij}^2 \leq 1$. So, given these, and the fact that $\sum_{i=1}^{d}\sum_{j=1}^{k} B_{ij}^2 = k$, we can assimilate a vector into the the right-hand side of the equation above to capture the characteristics just derived, providing an upper bound for the trace value

$$\text{trace}(\mathbf{U}^T\mathbf{M}\mathbf{U}) \leq \max_{b\in[0,1]^d : ||b|| \leq k} \sum_{i=1}^{d} \Lambda_{ii} b_i$$

The right-hand side of the equality above is simply a weighted sum of the d largest eigenvalues of matrix $\mathbf{M}$. Noting that the sum of elements of the vector b must be less than or equal to k and the eigenvalues are along the diagonal of $\mathbf{\Lambda}$ in a descending order by assumption without loss of generality, the maximum is achieved when the first k elements are set to one and the rest to zero, that is $\sum_{i=1}^{k} \Lambda_{ii}$.

Thus, for any matrix $\mathbf{U} \in \mathbb{R}^{d\times k}$ with orthonormal columns we have

$$\text{trace}(\mathbf{U}^T\mathbf{M}\mathbf{U}) \leq \sum_{i=1}^{k} \Lambda_{ii}$$

If we place $\mathbf{U}$ by the matrix whose columns are the k eigenvectors of $\mathbf{M}$, corresponding with the k biggest eigenvalues we get

$$\text{trace}(\underbrace{\mathbf{U}^T\mathbf{U}}_{I}\mathbf{\Lambda}) = \sum_{i=1}^{k} \Lambda_{ii}$$

Therefore, we observe that $\text{trace}(\mathbf{U}^T\mathbf{M}\mathbf{U}) = \sum_{i=1}^{k} \Lambda_{ii}$ when $\mathbf{U}$'s columns are the k leading eigenvectors of $\mathbf{M}$ and the proof is concluded. ■

The PCA algorithm is summarized in Algorithm 3. We construct $\mathbf{M}$ in time $O(Nd^2)$ and compute its eigendecomposition in $O(d^3)$ and so the computational complexity of PCA is $O(Nd^2 + d^3)$

Algorithm 3: PCA($\mathbf{x}_1, \ldots, \mathbf{x}_N, k$)

Input: N data points $\mathbf{x}_1, \ldots, \mathbf{x}_N$
number of principal components k
Output: k principal components $\mathbf{u}_1, \ldots, \mathbf{u}_k$
Compute $\mathbf{M} = \sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^T$
Compute k eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_k$ of $\mathbf{M}$ corresponding to the k largest eigenvalues subject to $\mathbf{u}_i^T \mathbf{u}_j = 0$ for all $i \neq j$ and $||\mathbf{u}_i|| = 1$ for all i
return $\mathbf{u}_1, \ldots, \mathbf{u}_k$;

2.7.2 Principle Component Analysis: Variance Maximization

We observed that PCA optimization problem was formed out of minimizing the reconstruction error in the previous section. However, an alternative approach comes into light when we attempt to decipher what the maximization problem in Equation (13) signifies. We start by discerning what the matrix $\sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^T$ represents. We calculate the expectation of each dimension of the data points, that is $\boldsymbol{\mu} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i$, that is in $\mathbb{R}^d$. If we subtract $\boldsymbol{\mu}$ from each data point, the matrix becomes $\sum_{i=1}^{n} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T$. This new matrix divided by n yields the covariance matrix. Therefore, the original matrix $\sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^T$ is in fact the covariance matrix of the *centered* data points times constant n. Centering the data points is referred to the act of subtracting the mean from the data points so as to make the mean zero, which is a common practice to do prior any calculation. Now that we have established matrix $\mathbf{M}$ is in fact the covariance matrix of the data points, we begin to look at the PCA problem from a different perspective.

PCA can be formulated as finding a different coordinate system for the data in which the variance of the data is maximized. Let $\mathbf{x}_i \in \mathbb{R}^d$ be a data point and $\mathbf{u} \in \mathbb{R}^d$ a basis vector of the new coordinate system. The projection of $\mathbf{x}_i$ onto $\mathbf{u}$ is the their dot product $\mathbf{u}^T \mathbf{x}_i$ assuming $||\mathbf{u}|| = 1$. What we would like is for the variance of the projections of $\mathbf{x}_i$ for $i = 1, \ldots, N$ to be maximum. Recalling what variance mathematically is and assuming that the data points are centered, i.e. their mean is zero, the variance of the projections is as follows

$$\begin{aligned}
&\frac{1}{N}\sum_{i=1}^{N}\left(\mathbf{u}^T\mathbf{x}_i\right)^2 \\
&= \frac{1}{N}\sum_{i=1}^{N}\left(\mathbf{u}^T\mathbf{x}_i\right)\left(\mathbf{x}_i^T\mathbf{u}\right) \\
&= \mathbf{u}^T\frac{1}{N}\sum_{i=1}^{N}\left(\mathbf{x}_i\mathbf{x}_i^T\right)\mathbf{u}
\end{aligned}$$

we substitute in the matrix $\mathbf{U} \in \mathbb{R}^{d\times k}$ whose columns are the new orthonormal basis vectors, as well as the covariance matrix Σ in place of $\frac{1}{N}\sum_i^N\left(\mathbf{x}_i\mathbf{x}_i^T\right)$ to arrive at the following optimization problem

$$\underset{\mathbf{U}\in\mathbb{R}^{d\times k},\mathbf{U}^T\mathbf{U}=I}{\arg\max}\ \text{trace}\left(\mathbf{U}^T\Sigma\mathbf{U}\right) \tag{14}$$

Equation (14) chooses a new coordinate system of $k(k < d)$ dimensions spanned by columns of $\mathbf{U}$ such that the total variance of the projected data points, that is the sum of the variance of each dimension d, is maximized. We already observed in Theorem 4 that the solution to this optimization problem is given by first k eigenvectors, arranged in an ascending order, of what we now know as the covariance matrix.

The *Lagrange multipliers* is an alternative way to deduce the same conclusion. For the sake of simplification, we assume we would like to find the variance of projections onto a single vector $\mathbf{u}$. The constrained optimization method is thus

$$\text{maximize } \mathbf{u}^T\Sigma\mathbf{u} \text{ s.t. } ||\mathbf{u}||^2 = 1$$

We will see in Section 4 that this term is called the *Rayleigh quotient*—of the covariance matrix—and yet another perspective is introduced that deduces eigenvectors are solutions to maximizing/minimizing such terms.

We use the Lagrange multiplier λ to combine the constraint with the function to be maximized [14]

$$\text{maximize } f(\mathbf{u},\lambda) = \mathbf{u}^T\Sigma\mathbf{u} - \lambda(\mathbf{u}^T\mathbf{u} - 1)$$

We solve for $\mathbf{u}$ by finding the stationary point of the above function, that is differentiating with respect to $\mathbf{u}$ and equating it to zero

$$\begin{aligned}
\frac{\partial f(\mathbf{u},\lambda)}{\partial\mathbf{u}} &= 2\mathbf{u}^T\Sigma - 2\lambda\mathbf{u}^T = 0 \\
\mathbf{u}^T\Sigma &= \lambda\mathbf{u}^T \\
(\mathbf{u}^T\Sigma)^T &= (\lambda\mathbf{u}^T)^T \\
\Sigma\mathbf{u} &= \lambda\mathbf{u}
\end{aligned}$$

And we have therefor arrived at the eigenvector equation of the covariance matrix σ. Thus, for k biggest eigenvalues $\lambda_1 \geq \ldots \geq \lambda_k \geq 0$ of covariance matrix Σ, $\mathbf{u}_1$ gives the orientation of the largest variance, $\mathbf{u}_2$ gives the orientation of the largest variance orthogonal to $\mathbf{u}_1$ (the second largest variance), all the way to $\mathbf{u}_k$ that is orthogonal to all the other eigenvectors, achieving the least variance among.

Therefor, we validated that the variance maximization and reconstruction error minimization in PCA are mathematically equivalent.

3 Introduction To Graph Embedding

Graph embedding methods aim to learn vector representations, embeddings, for the vertices of a graph. The goal is to produce an embedding for each of the vertices of the graph in a way that capture the graph topology, node-to-node relationship, or some relevant information of interest regarding each nodes, the graph, or the subgraph. The geometric relations in the latent space, where nodes are projected into, correspond to relations among them (e.g. links) in the original graph [15].

The low-dimensional embeddings represent the high-dimensional non-Euclidean graph-structured data and can be subsequently used for downstream tasks, such as community detection and link prediction.

Goyal and Ferrara [16] abstract the applications of graph embeddings into four categories: node classification, link prediction, clustering, and visualization. So we may be interested in determining labels of vertices of a partially labeled graph [17], to predict missing links between vertices [18], or to cluster similar nodes together [19]. A multitude of methods exists for such applications. For node classification there are methods that extract features from the graph vertices to subsequently feed them into a classifier [20], or methods that exploit random walks to propagate the labels [21]. Among approaches for link prediction exist maximum likelihood models [22] or similarity measure [23].

3.1 Definition and Preliminaries

3.1.1 Definition of Graphs and some of their Associated Matrices

Let $G = (\mathcal{V}, E)$ be a graph that is a collection of vertices (nodes) represented by the set $\mathcal{V} = \{v_1, \dots, v_n\}$ and edges (links) among them represented by $E \subseteq \mathcal{V} \times \mathcal{V}$.

The *adjacency matrix* of a graph is denoted by $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ whose elements are 1 or 0, representing existence of edges among the vertices. More concretely $A_{ij} = 1$ if $(v_i, v_j) \in E$ otherwise $A_{ij} = 0$. For a *weighted* graph, however, a non-negative weight is associated with every edge in graph and the adjacency matrix is directly translated into a *weight matrix*—a.k.a. *weighted adjacency matrix*—$\mathbf{W}$ where $W_{ij} \geq 0 \quad \forall i, j \in [n]$. If nodes v_i and v_j are not connected then $W_{ij} = 0$

We assume that the graphs are *undirected*—unless stated otherwise—, that is if $(v_i, v_j) \in E$ then $(v_j, v_i) \in E$. In such graphs we may use the set notation $\{v_i, v_j\} \in E$ instead of the tuple. And the corresponding adjacency or weight matrices are symmetric, that is $W_{ij} = W_{ji} \quad \forall i, j \in [n]$.

The edge wights W_{ij} are generally studied as a measure of similarity between nodes v_i and v_j and the higher it is, the more similar the two nodes are presumed to be [16].

For an unweighted undirected graph G, the *degree* of a vertex v_i, denoted by d_i, is defined as follows

$$d_i = \sum_{j}^{n} A_{ij}$$

In case of a weight matrix this is directly translated into the following

$$d_i = \sum_{j}^{n} W_{ij}$$

This gives way to another matrix associated with graphs called the *degree matrix* **D**. The matrix **D** is a diagonal matrix that has the degrees $d_1, \ldots, d_n$ along its diagonal and 0 elsewhere.

3.1.2 Graph Embedding: An Encoder-decoder Framework

We try to put the concept of graph embedding[1] introduced thus far on a concrete and unified footing by shedding light on the *encoder-decoder* framework, proposed by Hamilton et al. [24].

Two key components of the framework are two mapping functions: an *encoder* and a *decoder*. The idea is for the model to be able to learn high-dimensional information about the graph—such information can be the structure of local graph neighborhoods of nodes, or classification labels for the nodes [17]—from the low-dimensional embeddings of nodes. To this end, the encoder should map each node to a low-dimensional vector, and the decoder should decode the structural information of the graph from the learned embeddings. In principle, the embeddings should suffice to provide all the needed information for the subsequent off-the-shelf machine learning algorithms. The *encoder* and *decoder* are functions as follows.

$$\text{ENC} : \mathcal{V} \rightarrow \mathbb{R}^d \tag{15}$$

$$\text{DEC} : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^+ \tag{16}$$

The encoder takes node $v_i \in \mathcal{V}$ as an input, and outputs the associated embedded vector $\mathbf{z}_i \in \mathbb{R}^d$. The decoder, on the other hand, takes two embeddings as inputs and outputs some similarity measure. This measure could be any user-defined similarity measure, e.g. the shortest path length between the two nodes [25] or the existence of an edge [26]. [24] claims that although numerous decoders are possible, in most instances in the literature decoders take the aforementioned *pairwise* form. In 3.2.3 we will see an approach where a *unary decoder* is used.

The course of action is therefore as follows. The encoder embeds input nodes of the original graph v_i and v_j and outputs the corresponding embeddings, $\mathbf{z}_i$ and $\mathbf{z}_j$ respectively. The two embedded vectors are then passed to the decoder. The decoder, thereafter, reconstructs a similarity measure between the pair of nodes v_i and v_j in the original graph. The problem is then turned into the optimization of the encoder-decoder model to minimize the error in the reconstruction so that

$$\text{DEC}(\text{ENC}(v_i), \text{ENC}(v_j)) \approx s_G(v_i, v_j) \tag{17}$$

[1]The term graph embedding is referred to the embeddings of the vertices of graphs throughout. However, in the literature, sub-graphs can also be embedded, as discussed in [24].

$s_G : \mathcal{V} \times \mathcal{V} \mapsto \mathbb{R}^+$ [2] is a similarity measure between the nodes of the graph and can thus be represented as a *similarity matrix* $\mathbf{S} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$. The similarity matrix is user-defined, examples of which include the adjacency matrix [26], $s_G(v_i, v_j) \triangleq \mathbf{A}_{i,j}$, or the probability of two nodes co-occurring in a fixed-length random walk over G [27, 28].

Many approaches of graph embedding differ in how they define similarity matrices. [16] states two different measures:

- *First-order proximity*

  Because edge weights provide the most primitive similarity measure between nodes, they are also named first-order proximity. An example of such weight is illustrated by Equation (20).

- *Second-order proximity*

  Second-order proximity provides measure of similarity of the neighborhoods of nodes. More specifically, if we denote row i of matrix $\mathbf{S}$ by $\mathbf{s}_i$, such vector would yield the similarity measure of vertex v_i with all other vertices, that is $\mathbf{s}_i = (S_{i1}, \ldots, S_{i|\mathcal{V}|})^T$. The second-order proximity of nodes v_i and v_j is thus given by the similarity of $\mathbf{s}_i$ and $\mathbf{s}_j$. Examples of higher-order proximity measures include *Common neighbors*, or *Katz Index*, etc. [29]. Techniques such as [27, 28] preserve high-order proximity among nodes.

The next building block is the bedrock of the learning paradigm: a user-defined loss function, $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, that measures the discrepancy between the estimated and the true similarity measure between all pairs of nodes. The empirical loss function is thus to be minimized over all the pairs of nodes in the training set $\mathcal{S}$

$$\mathcal{L} = \sum_{(v_i, v_j) \in \mathcal{S}} \ell(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j), s_G(v_i, v_j)) \tag{18}$$

Minimizing the loss function in Equation (18) results in a trained encoder-decoder model. The model can then be used to output embeddings for the nodes. The resulting latent features may be fed into downstream machine learning algorithms to, for instance, classify node labels or perform link predictions.

The four primary methodological components of graph embeddings discussed thus far can be summarised as follows.

1. A **similarity matrix, S**, that describes a notion of user-defined similarity between the nodes of graph G—or the neighborhoods of the nodes.

2. An **encoder function, ENC**, that encodes the nodes into latent vector representations, i.e. embeddings. Usually most of the parameters of the model that are parameters of the in the encoder function.

3. A **decoder function, DEC**, which estimates the pairwise similarity measures from the generated embeddings and usually has no trainable parameters [24].

[2] The subscript emphasizes that the function is applied to the vertices of graph G.

4. A **loss function** $\mathcal{L}$, that quantifies the quality of the reconstructed pairwise similarity measures with the help of true similarity measure function s_G.

The existing graph embeddings differ primarily in how they define these four components, two examples of which we will observe in Section 4 and Section 5

3.2 A Taxonomy on Algorithmic Approaches

Initially, graph embedding algorithms were developed as a means to reducing the dimensionality of the data [16]. A powerful approach is Laplacian eigenmaps method [13]—which we will extensively study in Section 4—where a graph is constructed that encodes some similarity notion among N d-dimensional data points. The data points are then embedded in a lower-dimensional vector space, preserving the proximity of the vertices of the graph. These methodologies however lack scalability given they often operate in time quadratic in dimensionality of the data $O(d^2)$.

More scalable methodologies lean more towards random walk based methods—of which Deepwalk [28] will be discussed in Section 5— or neural networks as studied in [30] that both captures the non-linear structure of the graph and leverages the sparsity of real-world graphs.

There has been a surge in survey papers on the topic, among which are [24, 16] that we will explore to present an overview of the existing taxonomy on graph embedding technique.

Goyal and Ferrara [16] categorize the existing graph embedding techniques into three broad categories: *factorization based* techniques, *random walk based* methods, and *deep learning based* approaches. Hamilton et al. [24] provide a slightly different taxonomy in that embedding can take place at two different scales of (1) embedding nodes or (2) embedding sub-graphs. Among the node embedding techniques, they introduce two categories of deep neural networks, and shallow embedding within which factorization based methods and random walk-based approaches fall.

3.2.1 Factorization based methods

Factorization based algorithms, inspired by classic dimensionality reduction techniques leverage the connection of graphs and matrices. More broadly, spectral graph theory aims to explore graphs through the lens of eigenvectors and eigenvalues of matrices naturally associated with graphs. These matrices, namely adjacency matrix, Laplacian matrix, etc. can be factorized to obtain node embeddings. The techniques used to factorize the representative matrices differ depending on the properties of the matrices. In Section 4 we will study an example of factorization-based method that leverages eigenvalue decomposition of the Laplacian matrix.

3.2.2 Random Walk based Methods

Random walks are exclusively useful when graphs are too large to be studied in their entirety. The sampled random walks from the graph are to approximate the structure of the whole graph and therefore prove to be scalable. Deepwalk, a proposed random

walk approach for node embedding by Perozzi et al. [28] will be explored closely in Section 5. A brief comparison will additionally be provided with another well-known approach named node2vec [27].

3.2.3 Deep Learning based Approaches

This branch of approaches has witnessed a surge in surveys recently [31, 32] and it is a growing research topic. We will not cover any algorithms in the arena, rather, we will provide a general overview and what lies ahead.

There are some drawbacks associated with factorization based and random walk based techniques [24]:

- As we will see, algorithms in factorization and random walk based techniques, e.g. algorithms in Section 4 and Section 5, do not leverage node attribute information if provided by the underlying network. Many real-world networks provide node features that are not represented by the graph, e.g. age, gender or average income for social networks, that might be of benefit if Incorporated in the learning process.

- Such techniques can be computationally inefficient since they grow linearly in number of graph vertices, $O(|\mathcal{V}|)$, and do not embrace parameter sharing as a means to regularization [3]. In short, parameter sharing reduces the number of parameters of the model which results in faster convergence and combats overfitting.

- These algorithms only generate embeddings for the nodes they are trained upon [33] and need extra rounds of optimization to produce embeddings for unseen nodes. This renders such algorithms incompetent for evolving or massive graphs.

Deep learning approaches were proposed to counter some or all of the aforementioned drawbacks. These approaches exploit more complex encoders inspired by neural networks, and leverage the structure and attributes of graphs, more so than previously-mentioned techniques.

Drawing on *auto-encoders*—a specific architecture of neural networks—that have been used for the purpose of dimensionality reduction [34], approaches DNGR [35], and SDNE [30] compress and reconstruct neighborhood information vectors of every node in graph. These compression and reconstruction are done through a deep auto-encoder as depicted in Figure 3, much like PCA discussed in Section 2.7. Many other approaches exist, inspired by different other neural network architectures [31].

There is additionally a rich research area that provides theoretical foundation determining how expressive graph neural networks are [36] and show how certain graph neural networks' expressiveness is akin to that of 1-dimensional Weisfeiler-Leman graph isomorphism heuristic (1-WL) [37].

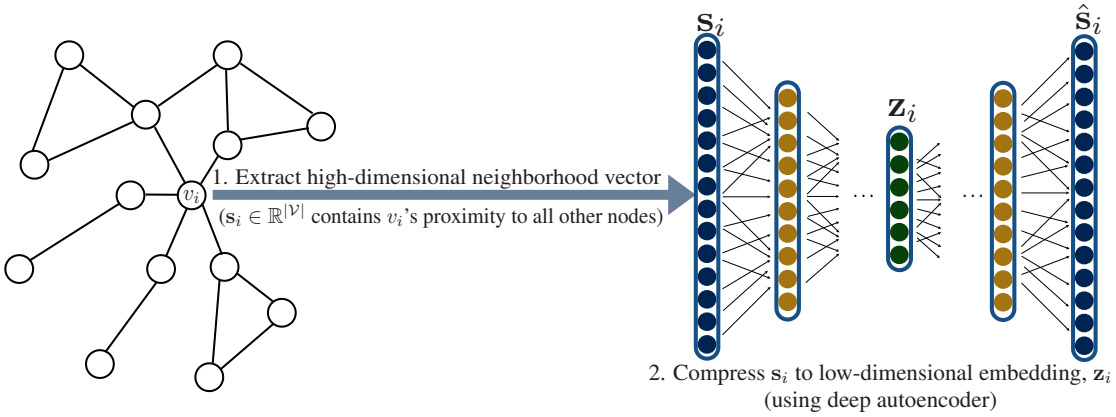

Figure 3: How neighborhood auto-encoder works in approaches SDNE and DNGR. Figure republished from [24] © 2017 IEEE.

4 Laplacian Eigenmaps: A Spectral Embedding Technique

Among the family of dimensionality reduction techniques—including linear dimensionality reduction method PCA, discussed in Subsection 2.7—there exists the family of *manifold learning* that gained traction due to their geometric intuition and non-linear nature. Manifold learning algorithms stem from the assumption that the input data is in a high-dimensional space $\mathbb{R}^d$ where in fact lies on or close to a k-dimensional manifold where $k \ll d$.

There is a natural correspondence between the graph Laplacian, the heat equation, and the Laplace-Beltrami operator on a manifold. Belkin and Niyogi [13] exploit such correspondence and offers a computationally efficient non-linear dimensionality reduction method that builds a representation for data points sampled from a low-dimensional manifold embedded in a high-dimensional space, while preserving the locality properties of the data points.

We first illustrate the core algorithm given by Belkin and Niyogi [13] and subsequently discuss its association to the geometric properties of the underlying manifold. We will see that the graph Laplacian obtained from the data in fact approximates the Laplace-Beltrami operator applied to the manifold. Furthermore, the connection between the graph Laplacian and the solution to the heat equation motivates [13] to select a particular set of weights for the constructed graph.

4.1 Laplacian Eigenmaps Algorithm

Laplacian eigenmaps algorithm is three-pronged:

1. Constructing the graph from the sampled data points.
2. Choosing weights for the edges of the graph.
3. Calculation of the eigenmaps with respect to the constructed graph.

The core algorithm is described in Algorithm 4. We furthermore dive into the some of the mathematical justifications that corroborate the intuition behind the laplacian eigenmaps, as well as a synopsis of the physics metaphor of the graph Laplacian and its properties.

Algorithm 4: Laplacian-eigenmaps($\mathbf{x}_1, \ldots, \mathbf{x}_N$)

Input: N data points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in $\mathbb{R}^d$
Output: N embedding vectors $\boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_N$ in $\mathbb{R}^m$
[*Constructing the graph G*] There is an edge between nodes i and j if $\mathbf{x}_i$ and $\mathbf{x}_j$ are "close" or "similar". This closeness can be defined in two ways:

- ϵ-neighborhood: Nodes i and j are connected by an edge if $||\mathbf{x}_i - \mathbf{x}_j||^2 < \epsilon$ for $\epsilon \in \mathbb{R}$
- n nearest neighbors: Nodes i and j are connected by an edge if $\mathbf{x}_j$ is among the n nearest neighbor of $\mathbf{x}_i$ or vice versa for some $n \in \mathbb{N}$.

[*Choosing the weights of G*] We choose weights for the edges of the constructed graph and construct the weight matrix. Again, there are two approaches:

- Heat kernel: Assign the following weight to the edge between nodes i and j:

$$W_{ij} = e^{-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{t}} \qquad t \in \mathbb{R}$$

- Adjacency matrix: $W_{ij} = 1$ if there is an edge between nodes i and j, otherwise $W_{ij} = 0$

[*Compute eigenmaps*] It is assumed the constructed graph G is connected. If not, the following step is applied to each connected component.
Compute the eigenvectors and the corresponding eigenvalues for the graph Laplacian of the graph G

$$\mathbf{L}\boldsymbol{\psi} = \lambda \mathbf{D}\boldsymbol{\psi} \tag{19}$$

Where $\mathbf{D}$ is a diagonal matrix whose elements take form of $D_{ii} = \sum_j W_{ij}$. Additionally, $\mathbf{L} = \mathbf{D} - \mathbf{W}$ is the Laplacian matrix. Let $\boldsymbol{\psi}_0, \ldots, \boldsymbol{\psi}_{k-1}$ be the solutions—which constitute the eigenvectors of the graph Laplacian—to Equation (19) in an ascending order with respect to their corresponding eigenvalues ($\boldsymbol{\psi}_0$ has the smallest eigenvalue).
Hence, the embedding vector of $\mathbf{x}_i$ is $\boldsymbol{\psi}_i \in \mathbb{R}^m$ whose elements are $(\boldsymbol{\psi}_1(i), \ldots, \boldsymbol{\psi}_m(i))$.
return $\boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_N$;

4.1.1 Similarity Graphs

Similarity graphs are graphs that attempt to capture local neighborhood relationships among a given set of data points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ with some notion of pairwise similarity. Some of the most well-known constructions for such transformations are what follows, as stated by [38]. We will also see one of such approaches in use in Section 4.

- *The ϵ-neighborhood graph*

  Each pair of vertices v_i and v_j are connected by an edge if the geodesic distance (e.g. L_2 distance) of the corresponding data points $\mathbf{x}_i$ and $\mathbf{x}_j$ is smaller than some $\epsilon > 0$. By definition, the distances between every endpoints of all the edges in the constructed graph are roughly of the same scale (at most ϵ), assigning additional weights to the edges of the graph would not supply more information. Therefore, the ϵ- neighborhood graphs are usually deemed unweighted.

- *The k-nearest neighbors graph*

  In this scheme, vertices v_i and v_j are connected if both vertices are among the k nearest neighbors of the other. The k nearest neighbor of a data point is the collection of k other data points that have the smallest distances to the original data point. The resulting graph is a directed graph since we established, by definition, an edge is placed if both of the corresponding data points are among each other's k nearest neighbors. Otherwise, we would be left with a directed graph since the neighborhood relationship is not symmetric. Having placed all the edges in either case, weights are assigned to them proportionately with the similarity, e.g. the inverse of the distance, of their endpoints.

- *The fully-connected graph*

  To constructing such graphs, all the vertices are connected and the weights are assigned to all edges to capture the local neighborhood relationships, that is the neighbors of each vertex are weighted. For this type of graphs, a similarity function—also known as the *kernel function* [39]—needs to be defined that quantifies the similarity between a pair of data points. And example of such a function is the following known as the Gaussian function

  $$f(\mathbf{x}_i, \mathbf{x}_j) = e^{-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2\sigma^2}} \tag{20}$$

  where the parameter σ controls the width of the neighborhoods. Therefore, all the edges of the graph are assigned weights computed by the Gaussian similarity function in Equation (20).

  The kernel function in Equation (20) decays with distance. Intuitively, the smaller the distance between two data points, the larger the weight of their connecting edge. Equation (20) is, in fact, the probability density function of a Gaussian random variable, in which σ represents the standard deviation of the distribution. Roughly, if we assume $\mathbf{x}_j$ to be a query point, the edge connecting it to data point $\mathbf{x}_i$ will be weighted heavily if $\mathbf{x}_j$ falls within distance σ of the data point $\mathbf{x}_i$.

4.2 The Graph Laplacian

4.2.1 Formal Definition of the Graph Laplacian

The *graph Laplacian*, the Laplacian matrix of graph $G = (E, \mathcal{V})$, is formally defined as the difference between the degree matrix and the adjacency matrix, that is $\mathbf{L} = \mathbf{D} - \mathbf{A}$. This entails that the Laplacian matrix has the degrees of the nodes on its diagonal, and -1 where there is an edge between nodes i and j and 0 otherwise. The elements of the Laplacian matrix, written out in full, are

$$L_{ij} = \begin{cases} d_i & \text{if } i = j, \\ -1 & \text{if } (i,j) \in E \text{ and } i \neq j, \\ 0 & \text{otherwise.} \end{cases} \tag{21}$$

A brief description of the physics intuition behind the Laplacian matrix (the graph Laplacian) is in order, to better grasp the idea behind Laplacian eigenmaps. We start by briefly formulating the *heat equation* both mathematically and intuitively and proceed to lay foundation from a physics standpoint. This allows to observe how the graph Laplacian is associated with the heat equation and its solution.

4.2.2 Heat Equation, Heat Kernel, and The Graph Laplacian

Diffusion is the analogous physical process that offers ample intuition for the graph Laplacian. Diffusion can be described as a process by which heat (or a fluid) moves from regions of high temperature (density for fluids) to regions of low. To exemplify, we assume that we have a simple undirected graph at our disposal where each node has a different temperature. The heat will flow in the direction of high temperatures to low, eventually reaching an approximate equilibrium. However, before equilibrium, the heat is distributed in such a way that the more closely connected nodes would have similar temperatures [40]. The nodes of the graphs can therefore be represented by a single value of temperature, embedding the nodes in a 1-dimensional vector space. It can be deduced that nodes that are closer in the graph have more similar temperatures and vice versa.

The mathematical formulation of the diffusion process holds significance in cementing that of the graph Laplacian. So, what follows is the mathematical modelling of diffusion in a simple undirected graph given by Newman [2]:

Suppose we have some substance at the vertices of a graph, denoted by ψ_i at vertex i. The substance flows along the edges, from node j to the adjacent node i at a rate $\alpha(\psi_j - \psi_i)$ where α is a constant named the *diffusion constant*. The amount of substance flowing from node j to i in a small interval of time is thus $\alpha(\psi_j - \psi_i)\mathrm{d}t$. The rate of change of ψ_i is therefore

$$\frac{\mathrm{d}\psi_i}{\mathrm{d}t} = \alpha \sum_j A_{ij}(\psi_j - \psi_i) \tag{22}$$

If we split the two terms we get

$$\frac{\mathrm{d}\psi_i}{\mathrm{d}t} = \alpha \sum_j A_{ij}\psi_j - \alpha\psi_i \sum_j A_{ij} = \alpha \sum_j A_{ij}\psi_j - \alpha\psi_i d_i \tag{23}$$

The expression that is being summed on the most right-hand side of Equation (23) can be divided into two constituents where $i = j$ and $i \neq j$ as follows.

$$\alpha \sum_j A_{ij}\psi_j = \alpha(\sum_{j,i\neq j} A_{ij}\psi_j + \sum_{j,i=j} A_{ij}\psi_j)$$

Given that the graph is assumed to be simple we have

$$\sum_{j,i=j} A_{ij}\psi_j = 0$$

Plugging this result back into Equation (23) we get

$$\frac{\mathrm{d}\psi_i}{\mathrm{d}t} = \alpha \sum_{j,i\neq j} A_{ij}\psi_j - \alpha\psi_i d_i$$

This allows us to incorporate the Kronecker delta δ_{ij} which is 1 if $i = j$ and 0 otherwise, yielding

$$\frac{\mathrm{d}\psi_i}{\mathrm{d}t} = \alpha \sum_j (A_{ij} - \delta_{ij} d_i)\psi_j \tag{24}$$

Equation (24) can be more concisely written in matrix form as follows

$$\frac{\mathrm{d}\boldsymbol{\psi}}{\mathrm{d}t} = \alpha(\mathbf{A} - \mathbf{D})\boldsymbol{\psi} \tag{25}$$

Where $\mathbf{A}$ is the adjacency matrix, $\mathbf{D}$ is the diagonal matrix that has the vertex degrees along its diagonal, and $\boldsymbol{\psi}$ is a column vector whose ith element is ψ_i.

We can move the right-hand side to the left; and factoring in the minus sign developed by this displacement yields a the matrix commonly defined as the graph Laplacian

$$\mathbf{L} = \mathbf{D} - \mathbf{A} \tag{26}$$

So the final diffusion equation becomes

$$\frac{\mathrm{d}\boldsymbol{\psi}}{\mathrm{d}t} + \alpha\mathbf{L}\boldsymbol{\psi} = 0 \tag{27}$$

The heat equation, as described in 4.2.3, takes the same form as Equation (27) with the matrix $\mathbf{L}$ replaced by the Laplacian operator ∇^2. Hence the name *graph Laplacian* for the matrix $\mathbf{L}$ [2].

We can solve Equation (27) by expanding vector $\boldsymbol{\psi}$ in the basis of the eigenvectors of $\mathbf{L}$. Writing the vector $\boldsymbol{\psi}$ as a linear combination of the eigenvector $\mathbf{u}_i$ of the Laplacian we get

$$\boldsymbol{\psi}(t) = \sum_i c_i(t)\mathbf{u}_i$$

where the coefficients $c_i(t)$ are functions of time. We substitute this form into Equation (27) and make use of the fact that $\mathbf{L}\mathbf{u}_i = \lambda_i \mathbf{u}_i$ for eigenvector $\mathbf{u}_i$ with its corresponding eigenvalue λ_i to get

$$\begin{aligned}&\sum_i \left(\frac{dc_i}{dt}\mathbf{u}_i + \alpha c_i(t)\mathbf{L}\mathbf{u}_i\right)\\ &= \sum_i \left(\frac{dc_i}{dt} + \alpha\lambda_i c_i(t)\right)\mathbf{u}_i = 0\end{aligned}$$

Considering that the eigenvectors of a symmetric matrix such as the Laplacian are orthogonal, taking the dot product of Equation above with any eigenvector results in

$$\frac{dc_i}{dt} + \alpha\lambda_i c_i(t) = 0$$

This differential equation, where the derivative of the function is proportional to itself has the solution of the following form

$$c_i(t) = c_i(0)e^{-\alpha\lambda_i t} \quad \text{for all } i \text{ and initial condition } c_i(t) \tag{28}$$

Equation (28) is named the *heat kernel* and is the solution to the discrete *heat equation*. We observed when discussing about similarity graphs in Section 4.1.1 how this continuous function would be an appropriate option for weight assigning to edges, which Algorithm 4 also exploits.

4.2.3 The Intuition Behind The Physical Heat Equation

Heat equation is a *partial differential equation* (PDE) that describes the course of heat distribution over time in a physical medium. For a multi-variable function $\boldsymbol{\psi}(x, y, z, t)$ where t denotes time and (x, y, z) denote the spatial variables, the heat equation is as follows

$$\frac{\partial\boldsymbol{\psi}}{\partial t} = \alpha\left(\frac{\partial^2\boldsymbol{\psi}}{\partial x^2} + \frac{\partial^2\boldsymbol{\psi}}{\partial y^2} + \frac{\partial^2\boldsymbol{\psi}}{\partial z^2}\right)$$

Written more compactly we have

$$\frac{\partial\boldsymbol{\psi}}{\partial t} = \alpha\nabla^2\boldsymbol{\psi} \tag{29}$$

Where ∇^2 is the *Laplace operator*, alse denoted by Δ, which is defined as the *divergence* of the gradient of a function on Euclidean space. Assuming n-dimensional vector space and a twice-differentiable real-valued function f we can formally write

$$\nabla^2 f := div \nabla f \tag{30}$$

Where the divergence operator is a vector operator that operates on a vector field. For a continuously differentiable vector field $\mathbf{F}$ in n-dimensional vector space, to calculate the divergence we simply compute the dot product of the gradient and the vector field

$$div\mathbf{F} = \nabla.\mathbf{F} = (\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n}).(F_{x_1}, \ldots, F_{x_n}) = \sum_i^n \frac{\partial F_{x_i}}{\partial x_i}$$

Intuitively the Laplace operator provides a measure of how different a value at a particular point is from the average of its neighbors. In the case of the heat equation, imagine two rods of different temperatures are connected at the ends. For a particular point on one of the rods, if the average temperature of its neighbouring points is higher than its temperature, the temperature of the point will rise. And in fact, the higher this difference, the faster the point's temperature increases. This proportionality directly, known as Newton's law of cooling, results in the heat equation.

4.2.4 Eigenvalues And Eigenvectors as Solutions to Optimization Problems

In our discussion about the linear dimensionality reduction PCA in Section 2.7 we observed that the eigenvectors associated with the largest eigenvalues of the covariance matrix provided a lower-dimensional coordinate system that could well describe the higher-dimensional data points. In the case of the Laplacian matrix, however, it turns out that the smallest eigenvalues and their associated eigenvectors deliver useful information about the underlying graph, as well as embed the vertices of the graph in a vector space.

What follows is the description of the *Rayleigh quotient* and how they pertain to eigenvalues and eigenvectors [41].

Lemma 5 *Let matrix* $\mathbf{M}$ *be symmetric with orthonormal basis of eigenvectors* $\mathbf{u}_1, \ldots, \mathbf{u}_n$ *corresponding to the eigenvalues* $\lambda_1, \ldots, \lambda_n$*. And let* $\mathbf{x}$ *be an arbitrary vector that can be written as*

$$\mathbf{x} = \sum_{i=1}^{n} c_i \mathbf{u}_i \qquad where\ c_i = \mathbf{u}_i^T \mathbf{x}$$

Then,

$$\mathbf{x}^T \mathbf{M} \mathbf{x} = \sum_{i=1}^{n} c_i^2 \lambda_i$$

Proof: We compute the Laplacian quadratic form of $\mathbf{x}$, plugging in the expansion

of $\mathbf{x}$ in the eigenbasis

$$\begin{aligned}\mathbf{x}^T\mathbf{M}\mathbf{x} &= \left(\sum_i c_i\mathbf{u}_i\right)^T \mathbf{M}\left(\sum_i c_i\mathbf{u}_i\right) \\ &= \left(\sum_i c_i\mathbf{u}_i\right)^T \left(\sum_i c_i\lambda_i\mathbf{u}_i\right) \\ &= \sum_{i,j} c_i c_j \lambda_i \mathbf{u}_i^T\mathbf{u}_j\end{aligned}$$

When $i = j$ the expression would evaluate to zero since the two distinct eigenvectors are orthogonal. So when $i \neq j$ we have

$$= \sum_{i=1}^{n} c_i^2\lambda_i$$

And the proof is concluded ■

To clarify how eigenvalues are solutions to quadratic optimization problems, we first define the *Rayleigh quotient* of a vector.

The Rayleigh quotient of a vector $\mathbf{x}$ with respect to a matrix $\mathbf{M}$ is defined as

$$\frac{\mathbf{x}^T\mathbf{M}\mathbf{x}}{\mathbf{x}^T\mathbf{x}} \tag{31}$$

Therefore, we can deduce that the Rayleigh quotient of an eigenvector with respect to matrix $\mathbf{M}$ is equal to its corresponding eigenvalue. So, if we have $\mathbf{M}\mathbf{u} = \lambda\mathbf{u}$ then

$$\frac{\mathbf{u}^T\mathbf{M}\mathbf{u}}{\mathbf{u}^T\mathbf{u}} = \frac{\mathbf{u}^T\lambda\mathbf{u}}{\mathbf{u}^T\mathbf{u}} = \lambda$$

The Courant-Fischer Theorem—a.k.a the min-max theorem—cements the statement we made above: finding the eigenvalues of a matrix is an optimization problem at heart. The theorem states that the vector that maximizes Equation (31) is the eigenvector corresponding to the largest eigenvalue of the symmetric matrix $\mathbf{M}$, and in fact, provides a similar characterization of all the eigenvalues of $\mathbf{M}$, implying it provides bounds on the Rayleigh quotient of any non-zero vector $\mathbf{x}$.

Theorem 6 (Courant-Fischer Theorem) *Let $\mathbf{M}$ be a symmetric matrix and suppose its eigenvalues are denoted by $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$ with the corresponding orthonormal eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_n$. Then the following holds*

$$\lambda_k = \max_{\substack{S\subseteq\mathbb{R}^n \\ dim(S)=k}} \min_{\substack{\mathbf{x}\in S \\ \mathbf{x}\neq\mathbf{0}}} \frac{\mathbf{x}^T\mathbf{M}\mathbf{x}}{\mathbf{x}^T\mathbf{x}} = \min_{\substack{T\subseteq\mathbb{R}^n \\ dim(T)=n-k+1}} \max_{\substack{\mathbf{x}\in T \\ \mathbf{x}\neq\mathbf{0}}} \frac{\mathbf{x}^T\mathbf{M}\mathbf{x}}{\mathbf{x}^T\mathbf{x}}$$

Proof: Given that $\mathbf{M}$ is a real-valued symmetric matrix, it has a set of orthonormal basis of eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_n$, corresponding to eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$

ordered in a descending order. Suppose the subspace S is span$\{\mathbf{u}_1, \ldots, \mathbf{u}_k\}$. So any vector $\mathbf{x}$ in such a subspace can be re-written as $\mathbf{x} = \sum_{i=1}^{k} c_i \mathbf{u}_i$. If we plug in the result of Lemma 5 we achieve

$$\frac{\mathbf{x}^T \mathbf{M} \mathbf{x}}{\mathbf{x}^T \mathbf{x}} = \frac{\sum_{i=1}^{k} c_i^2 \lambda_i}{\sum_{i=1}^{k} c_i^2} \geq \frac{\sum_{i=1}^{k} c_i^2 \lambda_k}{\sum_{i=1}^{k} c_i^2}$$

So this proves that λ_k is the minimum of the Rayleigh quotient. We subsequently need to prove that this is also the maximum. Assuming the subspace T is span$\{\mathbf{u}_k, \ldots, \mathbf{u}_n\}$. Considering the subspaces T and S of dimensions $n - k + 1, k$ respectively, it can be seen their intersection is non-empty. Consequently, there exists a vector in their intersection that can be written as

$$\mathbf{x} = \sum_{i=k}^{n} c_i \mathbf{u}_i$$

so

$$\frac{\mathbf{x}^T \mathbf{M} \mathbf{x}}{\mathbf{x}^T \mathbf{x}} = \frac{\sum_{i=k}^{n} c_i^2 \lambda_i}{\sum_{i=k}^{n} c_i^2} \leq \frac{\sum_{i=k}^{n} c_i^2 \lambda_k}{\sum_{i=k}^{n} c_i^2}$$

And this concludes the proof.

■

Thus in a more general sense we can state the following theorem.

Theorem 7 *Let $\mathbf{M} \in \mathbb{R}^{n \times n}$ be a symmetric matrix and suppose its eigenvalues are denoted by $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$ associated with orthonormal eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_n$. Then for an arbitrary vector $\mathbf{x}$ we have*

$$\max_{\mathbf{x} \neq \mathbf{0}} \frac{\mathbf{x}^T \mathbf{M} \mathbf{x}}{\mathbf{x}^T \mathbf{x}} = \lambda_1 \text{ where } \mathbf{u}_1 \in \arg\max \frac{\mathbf{x}^T \mathbf{M} \mathbf{x}}{\mathbf{x}^T \mathbf{x}}$$
$$\min_{\mathbf{x} \neq \mathbf{0}} \frac{\mathbf{x}^T \mathbf{M} \mathbf{x}}{\mathbf{x}^T \mathbf{x}} = \lambda_n \text{ where } \mathbf{u}_n \in \arg\max \frac{\mathbf{x}^T \mathbf{M} \mathbf{x}}{\mathbf{x}^T \mathbf{x}}$$

Proof: There are numerous ways to prove, such as using the spectral decomposition of $\mathbf{M}$. The easiest way however is to use Lemma 5.

Applying Lemma 5 we get

$$\frac{\mathbf{x}^T \mathbf{M} \mathbf{x}}{\mathbf{x}^T \mathbf{x}} = \frac{\sum_{i=1}^{n} c_i^2 \lambda_i}{\sum_{i=1}^{n} c_i^2} \leq \frac{\sum_{i=1}^{n} c_i^2 \lambda_1}{\sum_{i=1}^{n} c_i^2} = \lambda_1$$

In a similar fashion we provide the lower bound

$$\frac{\mathbf{x}^T \mathbf{M} \mathbf{x}}{\mathbf{x}^T \mathbf{x}} = \frac{\sum_{i=1}^{n} c_i^2 \lambda_i}{\sum_{i=1}^{n} c_i^2} \geq \frac{\sum_{i=1}^{n} c_i^2 \lambda_n}{\sum_{i=1}^{n} c_i^2} = \lambda_n$$

It is not hard to observe that the equality is achieved when vector $\mathbf{x}$ is the corresponding eigenvector.

■

The characterization provided by Theorem 7 can be generalized further to all eigenvalues of $\mathbf{M}$

$$\mathbf{u}_i \in \underset{\mathbf{x}^T\mathbf{u}_j=0:j<i}{\arg\max} \frac{\mathbf{x}^T\mathbf{M}\mathbf{x}}{\mathbf{x}^T\mathbf{x}} \quad \text{for all } 2 \leq i \leq n \tag{32}$$

We will not closely study the proof here but it follows Theorem 7 by induction.

4.3 Mathematical Justification and Intuition

The premise of Laplacian eigenmaps is that given N data points we construct an undirected weighted graph $G = (\mathcal{V}, E, w), w : E \longrightarrow \mathbb{R}^+$ with some predefined notion of how edges are drawn between vertices as seen in Algorithm 4. It is more intuitive to first consider the problem of mapping the graph to a line, that is embed the vertices of the graph in a 1-dimensional space. The goal in this case would be that more strongly connected nodes in the graph, be mapped to closer points on the line. This directly translates into minimizing the following expression

$$\sum_{i,j\in\mathcal{V}} W_{ij}(\psi_i - \psi_j)^2 \tag{33}$$

Where $\psi_i \in \mathbb{R}$ is the mapped 1-dimensional coordinate we would like to optimize. This expression is known as the *Laplacian quadratic* form [41]. Using the fact that $D_{ii} = \sum_j W_{ij}$ and $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{W}$ we can expand Equation (33) as follows

$$\begin{aligned}
&\sum_{i,j} W_{ij}(\psi_i - \psi_j)^2 = \sum_{i,j} W_{ij}(\psi_i^2 + \psi_j^2 - 2\psi_i\psi_j) \\
&= \sum_{i,j} \psi_i^2 W_{ij} + \sum_{i,j} \psi_j^2 W_{ij} - 2\sum_{i,j} \psi_i\psi_j W_{ij} \\
&= \sum_{i} \psi_i^2 D_{ii} + \sum_{j} \psi_j^2 D_{jj} - 2\sum_{i,j} \psi_i\psi_j W_{ij} \\
&= 2(\boldsymbol{\psi}^T\mathbf{D}\boldsymbol{\psi} - \boldsymbol{\psi}^T\mathbf{W}\boldsymbol{\psi}) \\
&= 2\boldsymbol{\psi}^T(\mathbf{D} - \mathbf{W})\psi \\
&= 2\boldsymbol{\psi}^T\mathbf{L}\boldsymbol{\psi}
\end{aligned}$$

where $\boldsymbol{\psi} = (\psi_1, \ldots, \psi_N)^T \in \mathbb{R}^N$, that is the 1-dimensional embedding for all the N vertices of the graph and $\boldsymbol{L}$ the Laplacian matrix.

We have thus shown that minimizing Equation (33) is equivalent to finding the following

$$\underset{\boldsymbol{\psi}^T\boldsymbol{D}\boldsymbol{\psi}=1}{\arg\min} \boldsymbol{\psi}^T\boldsymbol{L}\boldsymbol{\psi} \tag{34}$$

We showed previously in Theorem 7 that $\boldsymbol{\psi}$ that solves the optimization problem above is in fact the eigenvector of $\boldsymbol{L}$ associated with the smallest eigenvalue. A

natural choice for the embedding vector $\boldsymbol{\psi}$ is the diagonal of the degree matrix $\mathbf{D}$. To remove such solution, the constraint $\boldsymbol{\psi}^T \boldsymbol{D}\boldsymbol{\psi} = 1$ is added.

The Laplacian matrix is positive semi-definite, meaning its eigenvalues are non-negative—which can be deduced from the expansion of Equation (33) we just performed above. It is not hard to see that, in fact, the constant vector $\mathbf{1}$ whose elements are one is an eigenvector of the Laplacian with eigenvalue zero (every row of the Laplacian sums to zero). In the case of a connected graph, $\mathbf{1}$ is the only eigenvector whose eigenvalues are zero [42]. Therefore, the trivial solution where all the vertices are collapsed onto the real number one must be eliminated. This means the solution to the optimization problem above is given by the eigenvector associated with the smallest non-zero eigenvalue

$$\boldsymbol{\psi}_{opt} = \underset{\substack{\boldsymbol{\psi}^T \boldsymbol{D}\boldsymbol{\psi}=1 \\ \boldsymbol{\psi}^T \boldsymbol{D}\mathbf{1}=0}}{\arg\min}\, \boldsymbol{\psi}^T \boldsymbol{L}\boldsymbol{\psi} \tag{35}$$

Theorem 6 tells us $\boldsymbol{\psi}_{\text{opt}}$ is the eigenvector corresponding with the second smallest eigenvalue.

We can extrapolate this 1-dimensional case to find m-dimensional embeddings of the graph vertices. In fact, this problem is analogous to the PCA algorithm discussed in Subsection 2.7. Likewise, the goal is to find an m-dimensional coordinate system onto which the graph vertices will be projected, that is matrix $\boldsymbol{\Psi}$ with m N-dimensional columns whose ith row provides the m-dimensional embedding vector of the ith vertex of the graph.

Writing Equation (35) in matrix form we get

$$\boldsymbol{\Psi}_{opt} = \underset{\boldsymbol{\Psi}^T \boldsymbol{D}\boldsymbol{\Psi}=I}{\arg\min}\, \text{trace}(\boldsymbol{\Psi}^T \boldsymbol{D}\boldsymbol{\Psi}) \tag{36}$$

4.4 The Laplace-Beltrami Operator on Riemannian Manifolds

The original setting of the Laplacian eigenmaps techniques is that the data points lie on an underlying manifold $\mathcal{M}$. And the graph Laplacian is in fact the graph analogy of the Laplace-Beltrami operator on manifolds. For a smooth m-dimensional Riemannian manifold $\mathcal{M} \subset \mathbb{R}^d$—that is a manifold accompanied by a metric that allows for distances to be measured locally [43]—we consider the mapping function $f : \mathcal{M} \mapsto \mathbb{R}$. The function f directly translates into embedding the data points lying on $\mathcal{M}$ in a 1-dimensional vector space. We also defined the Laplace-Beltrami operator on Riemannian manifolds—in Equation (30)—to be the divergence of a vector field (∇f) in local coordinates of the manifold. Therefore, the change in output for a small δx can be approximated by the dot product of the gradient of f and the vector of change in the input variables of f

$$|f(x + \delta x) - f(x)| \approx |\nabla f(x).\delta x| \leq ||\nabla f|| \quad ||\delta x||$$

where the inequality holds by the Cauchy–Schwarz inequality. We can thus deduce the smaller the $||\nabla f||$, the smaller the change in the output of the mapping function. Consequently, the points close to the point x would also be mapped close to the mapping of x which is $f(x)$.

The problem hence amounts to finding a mapping function f that best maintains locality on all the data points lying on manifold $\mathcal{M}$ by minimizing the squared gradient of f

$$\underset{||f||_{L^2(\mathcal{M})}=1}{\arg\min} \int_{\mathcal{M}} ||\nabla f(x)||^2$$

This minimization is equivalent to minimizing the Laplace quadratic form in Equation (33) on a graph and in fact is the minimization of the Rayleigh quotient of the Laplace-Beltrami operator with respect to function f. Thus the minimization reduces to finding the eigenfunctions of the Laplace-Beltrami operator ∇^2 or Δ.

5 Random Walk Based Approches

Random walk-based approaches are staple of literature on graph embedding. As most algorithms have high levels of similarity we will discuss one, Deepwalk as presented by Perozzi et al. [28], in detail, then a comparison is made between that and node2vec, another heavily used method.

Deepwalk provides a novel approach in encoding the nodes of a graph in a continuous vector space. By modelling a stream of random walks, Deepwalk acquires *social representations* of the graph's nodes. The social representations are then encoded as latent features of the vertices that capture neighborhood similarity among nodes.

5.1 Problem Definition

Perozzi et al. [28] study the classification of members of a social network into one or multiple categories. Let $G = (\mathcal{V}, E)$ where $\mathcal{V}$ represents the members of the network whose connections is represented by $E \subseteq (\mathcal{V} \times \mathcal{V})$. Then, $G_L = (\mathcal{V}, E, \mathcal{X}, Y)$ is a partially labeled graph, where $\mathcal{X} \in \mathbb{R}^{|\mathcal{V}| \times k}$ with k being the size of the feature space for every representation vector and $Y \in \mathbb{R}^{|\mathcal{V}| \times \mathcal{Y}}$ where $\mathcal{Y}$ is the label set.

In a traditional machine learning setting, the goal is to find a classifier $h : \mathcal{X} \longrightarrow \mathcal{Y}$ that does not take into account the underlying connection among the data points that is the graph G. In the literature, this type of classification that classifies the unlabeled data based upon the labeled data points while also accounting for the underlying connections among the data points delineated by the structure of graph G is known as the *collective classification* problem [44]. Due to the fact that inferring the labels is an NP-hard problem, *iterative approximate inference algorithms* are used to compute the posterior distribution of labels [44].

Deepwalk, on the other hand, does not exploit the partially-labelled graph G_L and presents an unsupervised algorithm that learns an embedding matrix $\mathbf{\Phi} \in \mathbb{R}^{|\mathcal{V}| \times d}$, that can also be interpreted as a mapping function, that is $\boldsymbol{\phi} : |\mathcal{V}| \mapsto \mathbb{R}^d$, where d is the small number of dimensions of the latent space that the nodes are projected onto.

5.2 Social Representation Learning and Language Modelling

Deepwalk is substantially inspired by language modelling. For his reason, we first discuss the approaches used in the language modelling realm to lay the groundwork for Deepwalk, and additionally to justify why the techniques can be applied to social representation learning tasks.

5.2.1 Random Walks

A random walk on graph G is a stochastic process defined by the sequence of moves of a random walker between the vertices of G. Perozzi et al. [28] denote a random walk starting at vertex v_i by W_{v_i}. Such a random walker generates a walk such as

$W^1_{v_i}, W^2_{v_i}, \ldots, W^k_{v_i}$ where $W^{k+1}_{v_i}$ is a vertex chosen uniformly from the neighbors of $W^k_{v_i}$.

Random walks have proved useful in capturing local community structures, as they have been used as a similarity measure among nodes. Examples can be seen in the works of Pons and Latapy [45] where random walks have been utilized for the purpose of community detection, or that of Konstas et al. [46] where friendship relations in social networks are used to improve recommendation systems, by taking a random walk-based approach.

Alongside random walks' inherent ability to gain local community information, two other reasons are presented by Perozzi et al. [28] to further cement random walks' efficacy. First, the random walks can be parallelized given that they are generated for the graph nodes simultaneously and distinctively. And second, in case of a future change to the graph, the learned model can be iteratively updated by sampling new random walks from the changed portion of the graph in a nearly linear time in the number of graph nodes.

5.2.2 Language Modelling and their Analogy with Random Walks on Graphs

Most often, the objective of a language modelling algorithm is to categorize documents by topic and a way to accomplish this is by grouping similar words that best categorize documents on the basis of their topics. For instance, in a passage about animals, we can safely presume that words such as "mammals" and "animals" are highly likely to occur. An intuitive way to model such a problem is to think in probabilistic terms, that is, maximization of the probability of the corpus.

The Skip-gram model introduced by Mikolov et al. [47], which lies at the heart of Deepwalk, generates word embeddings to capture semantic and syntactic relationship among words in that similar words will also have similar word representation vectors. A well-known extension of the Skip-gram model was introduced by Mikolov et al. [48] that enhanced both the quality and the speed of the original skip-gram by introducing the concept of window.

Skip-gram's objective in Mikolov et al. [47] is to provide word embeddings that prove beneficial for predicting the words in a specific-sized window within the word. Formally, for a training sequence of words $(w_1, w_2, \ldots, w_n)$, Skip-gram aims to tune a set of parameters $\mathbf{\Phi}$ such that probability of the sequence of words, that is the product of independent conditional probabilities of each word in the context (the window) of every word of the sentence or the *target word*, is maximized

$$\arg\max_{\mathbf{\Phi}} \frac{1}{n} \prod_{t=1}^{n} \prod_{-c \leq j \leq c, j \neq 0} \mathbb{P}[w_{t+j} | w_t; \mathbf{\Phi}]$$

Taking the logarithm of the above equation, [48] re-defines the objective function of Skip-gram as a maximization of the average log probability

$$\frac{1}{n} \sum_{t=1}^{n} \sum_{-c \leq j \leq c, j \neq 0} \log \mathbb{P}[w_{t+j} | w_t] \tag{37}$$

c is defined to be the size of the context of each word w_t, that is the size of the window. The higher the c is, the more training data we possess, which can lead to higher accuracy albeit probably leading to more training time. On a positive note, c removes the constraint of order, that is Skip-gram maximizes the probability of word's appearance within the context of the target word, without imposing any restriction of where in the context. In the simplest form, the probability term in expression (37) is the following, approximated by the softmax function

$$\mathbb{P}[w_i|w_j] = \frac{e^{\mathbf{\Phi}(w_i)^T\mathbf{\Phi}(w_j)}}{\sum_{w\in\mathcal{W}} e^{\mathbf{\Phi}(w)^T\mathbf{\Phi}(w_j)}} \tag{38}$$

Where $\mathcal{W}$ is the entire vocabulary and $\mathbf{\Phi}(w_j)$ and $\mathbf{\Phi}(w_i)$ are the embeddings of the words w_j and w_i respectively. The parameters that we would like to optimize by maximizing the conditional probabilities of the words in a word sequence is the embedding matrix $\mathbf{\Phi}$.

The normalizing term in the denominator of the conditional probability (38) can become very expensive as the vocabulary increases in size. This inefficiency is counteracted by introducing the notion of *hierarchical softmax* which will be extensively discussed in the following section.

The analog of the "word sequences" in language modelling is the "random walks" in Deepwalk where the objective appears as maximizing the likelihood of vertex v_i given all the previously visited vertices along the random walk, that is

$$\mathbb{P}[v_i|(v_1, v_2, \ldots, v_{i-1})]$$

The latent feature vectors need to be incorporated into above-mentioned conditional probability, just as they did in Skip-gram, since they constitute the parameters Deepwalk will attempt to tune. And so the conditional probability becomes

$$\mathbb{P}[v_i|(\mathbf{\Phi}(v_1), \mathbf{\Phi}(v_2), \ldots, \mathbf{\Phi}(v_{i-1})] \tag{39}$$

Considering all the aforementioned characteristics of Skip-gram and additionally the fact that Deepwalk aims to maximize likelihoods of the seen vertices in each sampled random walk, the following cost function is reasonably defined

$$\min_{\mathbf{\Phi}} \quad -\log \mathbb{P}[\{v_{i-w}, \ldots, v_{i+w}\}\backslash\{v_i\}|\mathbf{\Phi}(v_i)] \tag{40}$$

Minimizing Equation (40) yields continuous vector representations for vertices preserving neighborhood proximity, meaning nodes that reside in vicinity of one another will obtain similar embeddings. What follows is how Deepwalk obtains such representation, its benefits and shortcomings.

5.3 Deepwalk Algorithm

Presented here is the Deepwalk algorithm, alongside the skip-gram implementation tailored to the sampled random walks from the graph. The random walks are

Algorithm 5: Deepwalk(G, w, d, γ, t)

Input: graph $G = (\mathcal{V}, E)$
window size w
embedding size d
walk length t
number of walk per node γ
Output: matrix of node embeddings $\mathbf{\Phi}$
Initialize $\mathbf{\Phi}$
Build a binary Tree T from $\mathcal{V}$
for $i = 0$ *to* γ **do**
    $\mathcal{O} = \text{Shuffle}(\mathcal{V})$
    for $v_i \in \mathcal{O}$ **do**
        $W_{v_i} = \text{Randomwalk}(G, v_i, t)$
        $\text{Skipgram}(\mathbf{\Phi}, W_{v_i}, w)$
return $\mathbf{\Phi}$;

analogous to the training sentences in the language modelling scenario. The concept of corpus is akin to graph and the vertices of the graph constitute the vocabulary.

Algorithm 5 delineates the course of Deepwalk. Adhering to the sequence in Algorithm 5 , the embedding matrix is initialized and a binary tree is constructed with the graph nodes at the leaves. The details of the tree construction is discussed in Section 5.3.2.

Having performed the initialization and the binary tree construction, the algorithm enters a loop of sampling γ random walks per vertex. Within each random walk of length t, Algorithm 5 passes over all the nodes of the graph, generating a random walk W_{v_i} rooted at v_i. The sampled random walk is subsequently passed to the Skip-gram procedure described in Algorithm 6 to tune embedding of node v_i in accordance with both the random walk generated rooted at v_i and the objective function.

5.3.1 Skip-gram

Skip-gram is at the center of the optimization process of embedding matrix. Each random walk W_{v_i} rooted at v_i is passed to Skip-gram, along with a fixed window size and the embedding matrix $\mathbf{\Phi}$.

Algorithm 6: Skip-gram($\mathbf{\Phi}, w, W_{v_i}$)

Input: window size w
random walk W_{v_i}
embedding matrix $\mathbf{\Phi}$
Output: matrix of node embeddings $\mathbf{\Phi}$
for $v_j \in W_{v_i}$ **do**
  for $u_k \in W_{v_i}[j-w : j+w]$ **do**
    $J(\mathbf{\Phi}) = -\log \mathbb{P}[u_k | \mathbf{\Phi}(v_j)]$
    $\mathbf{\Phi} = \mathbf{\Phi} - \frac{\partial J(\mathbf{\Phi})}{\partial \mathbf{\Phi}}$
return $\mathbf{\Phi}$;

Skip-gram implementation of Deepwalk is inspired by that of language modelling, in the sense that it attempts to maximize co-occurrence probabilities of nodes in a random walk, within a fixed-size window. For every node v_j in random walk W_{v_i}, Skip-gram in Algorithm 6 iterates over all $2w$ neighbors of v_j, and maximizes each neighbor's probability given the current embedding of v_j, $\mathbf{\Phi}(v_j) \in \mathbb{R}^d$. This posterior probability is approximated by a softmax function, as illustrated in Equation (38). The normalizing term in the denominator passes through all the graph nodes—the time complexity of which is $O(|\mathcal{V}|)$—which can be expensive to compute. This shortcoming brings us to the following section.

5.3.2 Hierarchical Softmax

As discussed previously, the normalizing factor in the softmax function Equation (38) (the probability calculation of Skip-gram in Algorithm 6) is expensive to calculate

and therefore not computationally viable. Hierarchical softmax is, on the other hand, a computationally efficient approximation of the softmax function, initially introduced by Morin and Bengio [49] and used by Mikolov et al. [48] and Mnih and Hinton [50] in area of language modelling.

In Deepwalk, hierarchical softmax employs a binary tree and in so doing reduces the time complexity of the conditional probability $\mathbb{P}[u_k|\mathbf{\Phi}(v_j)]$ from $\mathcal{O}(|\mathcal{V}|)$ to $\mathcal{O}(\log|\mathcal{V}|)$. The structure of the binary tree delineates how the calculation of the posterior probability $\mathbb{P}[u_k|\mathbf{\Phi}(v_j)]$ takes place. If we assign the vertices of the graph to the leaves of the tree, the calculation of $\mathbb{P}[u_k|\mathbf{\Phi}(v_j)]$ can be revisited as maximizing specific paths in the tree that originate at the root of the tree and ends at the leaf corresponding to u_k. Mikolov et al. [48] provide the following concrete notation based on which hierarchical softmax defines the posterior probability $\mathbb{P}[u_k|\mathbf{\Phi}(v_j)]$. Let $n(u_k, i)$ be the i-th node on the path from the root to u_k and $ch(b)$ be a fixed child of node b. In addition, L denotes the length of the path and $[\![x]\!]$ is 1 if x is true and -1 otherwise. Thus, the conditional probability $\mathbb{P}[u_k|\mathbf{\Phi}(v_j)]$ can be re-written as

$$\mathbb{P}[u_k|\mathbf{\Phi}(v_j)] = \prod_{j=1}^{L-1} \sigma([\![n(u_k, j+1) = ch(n(u_k, j))]\!].\mathbf{\Phi}(v_j)^T z_{n(u_k,j)}) \tag{41}$$

where $z_{n(u_k,j)}$ is a vector representation of the node $n(u_k, j))$ and $\sigma(x) = \frac{1}{1+e^{-x}}$. The cost of computing $\mathbb{P}[u_k|\mathbf{\Phi}(v_j)]$, as seen in Equation (41), is proportional to the length of the paths leading to u_k, which renders it logarithmic in $|\mathcal{V}|$ ($\mathcal{O}(\log|\mathcal{V}|)$).Thus, hierarchical softmax formulation has a vector representation for each vertex of graph and one for every inner node of the binary tree. We are however only interested in the embedding vectors of the graph vertices and the optimized binary tree nodes' embeddings are solely a means to an end.

The set of parameters that are to be optimized are the embedding matrix $\mathbf{\Psi}$ and the embedding matrix of all inner nodes of the constructed binary tree $\mathbf{Z}$. Stochastic gradient descent [51] is the optimization method of choice and the the partial derivatives in Algorithm 6 are estimated using the back-propagation algorithm [10].

An overview of Deepwalk is depicted in Figure 4. The tree is in fact akin to a neural network where a single representation vector is passed through the layers, with the last layer having the sigmoid function as the activation function to output a probability distribution, as briefly studied in Section 2.6. viewed in the encoder-decoder light discussed in Section 3.1.2, Deepwalk uses the sigmoid function as the decoder to estimate probability; and the loss function is log-loss, studied in Section 2.5.1.

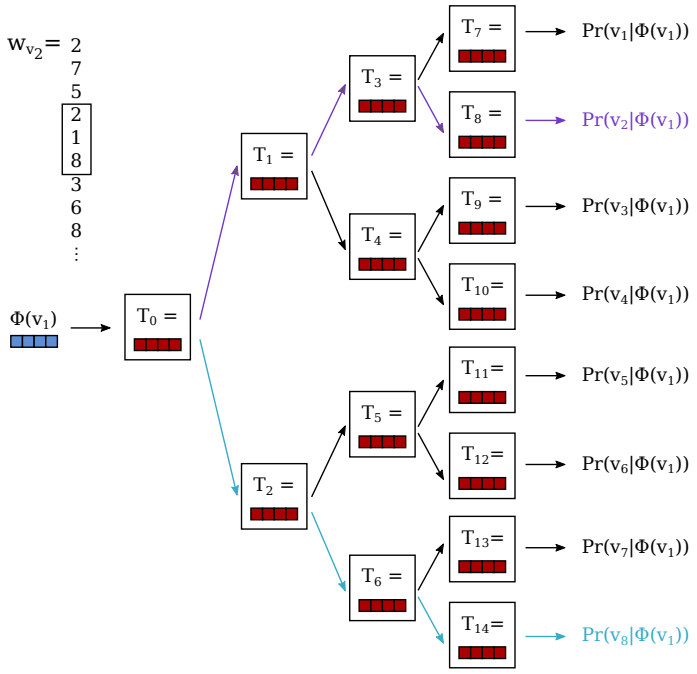

Figure 4: A random walk W_{v_2} is generated, rooted at vertex v_2 on a graph with eight vertices. A window of size three slides through the vertices. For example, the embedding of vertex v_1 is computed in such a way that the probability of v_1 co-occurring with its context $\{v_2, v_8\}$ is maximized. This amounts to maximizing the probabilities of the paths leading to vertices v_2 and v_8.

5.4 Comparison between Deepwalk and node2vec

A similar method called node2vec [27] uses *biased* random walks to generate representation vector for graph vertices. We will not scrutinize node2vec in same level of detail as of Deepwalk in this manuscript; however it is worthwhile to point out their similarities and differences briefly.

Deepwalk [28] and node2vec [27] are similar in that they both maximize the probability of node co-occurrences in sampled random walks across the graph. The difference however lies in how the random walks are sampled. The former uses unbiased random walks, that is subsequent vertices at each step in the random walk are chosen uniformly, whereas the latter biases the random walks using two random walk hyperparameters *return parameter* p and *in-out parameter* q. The return parameter p is responsible for controlling the likelihood of immediately revisiting a vertex; and the in-out parameter q controls the likelihood that the walk revisits a vertex's one-hop neighborhood. Deepwalk can in fact be viewed as a special case of node2vec where $p = q = 1$.

In the context of the encoder-decoder framework, node2vec utilizes the softmax function as its decoder function to approximate the probabilities of vertices co-occurring with vertices in their neighborhood. The loss function is again formulated as the log-loss used also by Deepwalk, followed by stochastic gradient descent as a means of optimization.

It was stipulated by Grover and Leskovec [27] that parameters p and q allowed tradeoff between the Depth-First Search and Bredth-First Search, which in turn implied tradeoff between learning embeddings that underpin local structural roles, or embeddings that reflect community structures.

6 Discussion

We started by laying foundation of machine learning algorithms in the hope of justifying the methodologies presented by the subsequent graph embedding algorithms, as well as drawing on the many similarities. The Laplacian eigenmaps in Section 4 provides a mathematically just algorithm for producing embeddings, although it does not constitute a scalable one given the mere construction of the Laplacian matrix is quadratic in number of graph nodes.

An algorithmically different method discussed in Section 5 is Deepwalk that is inspired by methods proposed in language modelling. Deepwalk is much scalable, linear in the number of graph vertices.

The sheer volume of massive data that are nowadays generated on a daily basis fuels the necessity of developing all the more scalable algorithms. On top of scalability, graph embedding techniques, although empirically successful, are lacking in some respects as well.

There is a stark lack of theoretical justification as to the learned embeddings are truly learned encoded graph information, and not mere exploitation of statistical features of the benchmarks on which they achieve state-of-the-art performance [24]. Although the mathematical justification of the factorization based method such as the Laplacian eigenmaps in Section 4 is legitimate, it is not however as just in the Deepwalk method studied in Section 5. There is much work to be done for rigorous mathematical justification as to decoder functions in Deepwalk (the sigmoid function) and node2vec (the softmax function) truly estimate co-occurrence probabilities. Moreover, the loss function used in both Deepwalk and node2vec is non-convex [52]; and optimization of non-convex functions is a highly developing area of research.

Most existing methods do not attempt to capture high-order structural motifs of graphs that play a key role in understanding how massive graphs function. To exemplify, exploitation of local graph motifs can enable graph convolutional neural networks to be applied to *directed* graphs as well [53].

The extension of this work is hoped to be a thorough study of the existing graph neural networks techniques and further theoretical study of their expressiveness and abilities on a unified mathematical setting across all literature. Given the correspondence between graph neural networks and artificial neural networks in the machine learning area, improvement in one can lead to exciting findings in the other. The theory of neural networks is a nascent research field that aims to provide insights into the tradeoff between the number of layers (depth) and the number of neurons per layer (width). Such theoretical findings can speak to neural networks' efficacy and in turn be beneficial for the growth of graph neural networks.

References

[1] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[2] Mark Newman. *Networks: an introduction*. Oxford university press, 2010.

[3] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[4] Sanjoy Dasgupta. *The hardness of k-means clustering*. Department of Computer Science and Engineering, University of California . . . , 2008.

[5] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[6] Albert B Novikoff. On convergence proofs for perceptrons. Technical report, STANFORD RESEARCH INST MENLO PARK CA, 1963.

[7] Dan Claudiu Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[8] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth annual conference of the international speech communication association*, 2014.

[9] John S Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing*, pages 227–236. Springer, 1990.

[10] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[11] Ali Ghodsi. Dimensionality reduction a short tutorial. *Department of Statistics and Actuarial Science, Univ. of Waterloo, Ontario, Canada*, 37:38, 2006.

[12] Sebastian Mika, Bernhard Schölkopf, Alex J Smola, Klaus-Robert Müller, Matthias Scholz, and Gunnar Rätsch. Kernel pca and de-noising in feature spaces. In *Advances in neural information processing systems*, pages 536–542, 1999.

[13] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Advances in neural information processing systems*, pages 585–591, 2002.

[14] Dimitri P Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.

[15] Peter D Hoff, Adrian E Raftery, and Mark S Handcock. Latent space approaches to social network analysis. *Journal of the american Statistical association*, 97(460):1090–1098, 2002.

[16] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[17] Smriti Bhagat, Graham Cormode, and S Muthukrishnan. Node classification in social networks. In *Social network data analytics*, pages 115–148. Springer, 2011.

[18] Seyed Mehran Kazemi and David Poole. Simple embedding for link prediction in knowledge graphs. In *Advances in Neural Information Processing Systems*, pages 4284–4295, 2018.

[19] Petros Drineas, Alan Frieze, Ravi Kannan, Santosh Vempala, and V Vinay. Clustering large graphs via the singular value decomposition. *Machine learning*, 56(1-3):9–33, 2004.

[20] Przemyslaw Kazienko and Tomasz Kajdanowicz. Label-dependent node classification in the network. *Neurocomputing*, 75(1):199–209, 2012.

[21] Arik Azran. The rendezvous algorithm: Multiclass semi-supervised learning with markov random walks. In *Proceedings of the 24th international conference on Machine learning*, pages 49–56. ACM, 2007.

[22] Aaron Clauset, Cristopher Moore, and Mark EJ Newman. Hierarchical structure and the prediction of missing links in networks. *Nature*, 453(7191):98, 2008.

[23] Panagiotis Symeonidis, Eleftherios Tiakas, and Yannis Manolopoulos. Transitive node similarity for link prediction in social networks with positive and negative links. In *Proceedings of the fourth ACM conference on Recommender systems*, pages 183–190. ACM, 2010.

[24] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[25] Andrey Kutuzov, Alexander Panchenko, Sarah Kohail, Mohammad Dorgham, Oleksiy Oliynyk, and Chris Biemann. Learning graph embeddings from wordnet-based similarity measures. *arXiv preprint arXiv:1808.05611*, 2018.

[26] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd international conference on World Wide Web*, pages 37–48. ACM, 2013.

[27] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[28] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[29] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114. ACM, 2016.

[30] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1225–1234. ACM, 2016.

[31] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[32] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[33] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[34] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[35] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Deep neural networks for learning graph representations. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[36] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[37] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019.

[38] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[39] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive data sets*. Cambridge university press, 2019.

[40] Laurenz Wiskott and Fabian Schönfeld. Laplacian matrix for dimensionality reduction and clustering. *arXiv preprint arXiv:1909.08381*, 2019.

[41] Daniel Spielman. Spectral graph theory. *Lecture Notes, Yale University*, pages 740–0776, 2009.

[42] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, NJ, 1996.

[43] Frank Schmidt. The laplace-beltrami-operator on riemannian manifolds. In *Seminar Shape Analysis*, 2014.

[44] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[45] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. In *International symposium on computer and information sciences*, pages 284–293. Springer, 2005.

[46] Ioannis Konstas, Vassilios Stathopoulos, and Joemon M Jose. On social networks and collaborative recommendation. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 195–202. ACM, 2009.

[47] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[48] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[49] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer, 2005.

[50] Andriy Mnih and Geoffrey E Hinton. A scalable hierarchical distributed language model. In *Advances in neural information processing systems*, pages 1081–1088, 2009.

[51] Léon Bottou. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8):12, 1991.

[52] Rene Vidal, Joan Bruna, Raja Giryes, and Stefano Soatto. Mathematics of deep learning. *arXiv preprint arXiv:1712.04741*, 2017.

[53] Federico Monti, Karl Otness, and Michael M Bronstein. Motifnet: a motif-based graph convolutional network for directed graphs. In *2018 IEEE Data Science Workshop (DSW)*, pages 225–228. IEEE, 2018.